```
// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// cache.hpp
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// ***************************************************************
// Copyright 1991-2003 iAnywhere Solutions, Inc.  All rights reserved.
// ***************************************************************
#ifndef CACHE_HPP
#define CACHE_HPP
class Cache;
class CacheChain {
public:
    CacheChain()
 : _head( NULL )
 , _refs( NULL )
    {
    }
    void LatchInfos() {
 _latch.get( _contention_counter( HASH_CONTENTION ) );
    }
    void UnlatchInfos() {
 _latch.give();
    }
    void Latch() {
 _mutex.get( _contention_counter( HASH_CONTENTION ) );
    }
    void Unlatch() {
 _mutex.give();
    }
    class PageInfo *Find( a_cache_name name );
    class PageInfo *AssuredFind( a_cache_name name );
    class PageInfo *FindOrInsert( a_cache_name name, PageInfo * insert );
    void Remove( PageInfo * remove );
    // Operations on cache refs
    void Add( class CacheRef * ref );
    a_bool Remove( class CacheRef * remove );
private:
    friend class Cache;
    friend class CacheRef;
    Mutex  _latch;
    Mutex  _mutex;
    class PageInfo  *_head;
    class CacheRef  *_refs;
};
typedef unsigned a_queue_time;
typedef a_byte a_prs_slot;
#define PRS_TOTAL_SLOTS 128
class DBSRVAPI PageInfo : public CacheInfo {
public:
    PageInfo( void * image );
    ~PageInfo();
```

```
    void PrepareForAdd( a_cache_name name, unsigned now );
    a_bool Scrub();
    void PrepareForRead( class IOCB * iocb );
    PageInfo * Reset( a_cache_name name, unsigned now );
    inline a_bool reapable_address_space() const; // and physical frame (addr is always mapped to something)
    inline a_bool reapable_physical_frame( class Cache *cm ) const;
    inline a_bool decommitable_image() const;
    a_bool do_read_to_write_lock( Worker *me );
    void do_write_to_read_lock( Worker *me );
    void write_lock();
    void add( class CacheRef *ref );
    void remove( class CacheRef *ref );
    void wait_for_lock( CacheChain *chain );
    a_bool is_dirty() const {
  return( _is.dirty );
    }
    void mark_dirty() {
  if( !is_dirty() ) {
      _file->mark_dirty( this );
  }
    }
    void do_write( PageInfo *delay );
    void write();
    void start_write( PageInfo *delay = NULL );
    void write_and_flush();
    void read();
    void w_complete();
    void r_complete();
    void remove_delay();
    void prep_for_read( Database *db );
    void map();
    void enqueue( class Cache *cache );
    inline void set_physmem_id( class Cache *cm, a_physmem_id physmem_id ); // only used in cachespt
    inline a_physmem_id get_physmem_id( class Cache *cm ) const;
    a_bool is_pending( void );
#if defined( AWE_CACHE )
    inline void map_to_physmem( class Cache *cm, a_physmem_id physmem_id );
#endif
private:
    PageInfo   *_next;  // on chain
    RWLatch  _latch;
    union a_name_reuse_state {
  struct {
      uint16 ref_count;
      uint16 is_in_hash;
  }  bits;
  uint32  as_cell;
    }   _state;
    union {
  a_byte  _flags;
```

```
 struct {
     a_byte dirty : 1;
     a_byte preimage_may_not_be_committed : 1;
     a_byte blob_page : 1;
     a_byte tracked_table_page : 1;
       a_byte      coldstart_recorded : 1;
 }  _is;
    };
    uint16  _tid;
    unsigned  _visited_at;
    a_prs_slot  _score;
    union { // only meaningful when _is.preimage_may_not_be_committed == TRUE
 a_physical_page _checkpoint_log_pos;
 a_page_id _rollback_page_no;
    };
    a_page_id  _real_page_no; // != page_no if mapped
    class IOCB * _pending;
    enum { // Overloaded values for _pending. Must have low bit on.
 REUSABLE = 1,
 JUST_INSTALLED = 3,
 IMAGE_MISSING = 5,
 PINNED = 7
    };
#if !PRODUCTION
    a_debug_count       first_lock;     // for debugging
    a_debug_count       first_write;    // for debugging
#endif
#ifdef CHECKSUM
    unsigned            checksum;
#endif
    // Unprotected by _latch.
    a_ptrint  _queue_slot;
public:
    a_bool read_to_write_lock();
    void write_to_read_lock();
    void unlock();
#if !PRODUCTION
    void verify();
#endif
public:
    a_bool HasName( a_cache_name name ) const {
 return( _name.as_uint == name.as_uint );
    }
    a_bool IsReusable() const {
 return( _pending == (IOCB *) REUSABLE );
    }
    a_bool IsMissingImage() const {
 return( _pending == (IOCB *) IMAGE_MISSING );
    }
    a_bool IsPinned() const {
```

```
 return( _pending == (IOCB *) PINNED );
   }
   a_bool IsDirty() const {
 return( _is.dirty );
   }
   a_bool IsInHash() const {
 return( _state.bits.is_in_hash );
   }
   a_bool IsAddressable() const {
 return( _image != NULL );
   }
   void MarkImageAsMissing() {
 _pending = (IOCB *) IMAGE_MISSING;
   }
   void MarkImageAsAvailable() {
 _pending = NULL;
   }
   a_bool Latch( class Cache * cm, a_cache_name name, a_bool shared = FALSE );
   void Latch( a_bool shared = FALSE ) {
 if( shared ) {
    _latch.get_shared();
 } else {
    _latch.get_exclusive();
 }
   }
   a_bool TryLatch( a_bool shared = FALSE ) {
 return( shared? _latch.try_get_shared() : _latch.try_get_exclusive() );
   }
   void ExclusiveToShared() {
 _latch.exclusive_to_shared();
   }
   a_bool SharedToExclusive() {
 return( _latch.shared_to_exclusive() );
   }
   a_bool TrySharedToExclusive() {
 return( _latch.try_shared_to_exclusive() );
   }
   a_bool IsLatched() const {
 return( _latch.have_latch() );
   }
   a_bool IsInUse() const {
 return( IsLatched() || IsPinned() );
   }
#if !PRODUCTION
   a_bool HaveExclusively() const {
 return( _latch.have_exclusive_latch() );
   }
#endif
   void Unlatch() {
 _latch.give();
```

```
    }
    void Unlock() {
  Unlatch();
    }
    void WaitForPending() {
  if( _pending ) {
      DoWaitForPending();
  }
    }
    void WaitForPreimage() {
  if( _is.preimage_may_not_be_committed ) {
      DoWaitForPreimage();
  }
    }
    void Shrink( Cache * cm );
    a_bool IsShrinkable();
    a_bool CanCommit();
    void Committed();
    void Pin();
    void Unpin();
    void WriteAndEvict();
    void Evict();
private:
    friend class Cache;
    friend class PageQueue;
    friend class CacheChain;
    friend class CacheRef;
    friend class CacheInfo;
    friend class InfoFriend;
    friend class DatabaseFile;
    friend class CheckpointLog;
    friend class IOCB;
    void RecordHit( class Cache * cm );
    a_bool CanPreventReuse();
    a_bool AllowReuse();
    a_bool SetInHash( a_bool in_hash );
    a_bool IsReapable() const;
    void Clean();
    void FinishLock( a_bool shared );
    void AssertNotFree( Database * db );
    class IOCB * StartRead();
    class IOCB * StartWrite( a_bool evict = FALSE );
    void SetPreImage( a_page_id page_no );
    void Write( a_bool wait = FALSE );
    void DoWaitForPending();
    void DoWaitForPreimage();
    a_bool JustInstalled() const {
  return _pending == (IOCB *) JUST_INSTALLED;
    }
    a_bool IsColdstartRecorded() const {
```

```
        return _is.coldstart_recorded;
    }
    void  SetColdstartRecorded( a_bool val ) {
        _is.coldstart_recorded = (a_byte) val;
    }
};
// _pending codes: iocb pointer
// 1 => reserved for local thread
// 2 => address space missing
// 3 => need data
// 4 => free
// 5 => reusable
// 6 => allocated for image
// A page_ref is in some chain iff page_no is not NULL_PAGE.
// The page pointed to by a page_ref is locked iff page is not NULL.
#define SUP_INDEX       65535
class DBSRVAPI CacheRef : public HeapObject {
public:
    CacheRef()
 : _chain( NULL ), _info( NULL ), _flags( 0 )
    {
 _name.as_member.page_no = NULL_PAGE;
    }
    CacheRef( a_database_number db_no, a_page_id page_no );
    CacheRef( a_database_number db_no, a_record_id const &id );
    CacheRef( PageInfo *info ) {
 force( info );
    }
    ~CacheRef() {
 releaseRefs();
    }
    CacheRef( CacheRef const &ref );
    CacheRef& operator= ( CacheRef const &ref );
    void Set( a_database_number db_no, a_record_id const &id );
    void Add( a_cache_name name, a_page_count index = 0 );
    void Remove();
    void Latch( a_bool shared = FALSE );
    void Latch( a_cache_name name, a_bool shared );
    void Lock( a_bool shared = FALSE ) {
 Latch( shared );
 mark_dirty( shared );
    }
    void Lock( a_cache_name name, a_bool shared = FALSE ) {
 Latch( name, shared );
 mark_dirty( shared );
    }
    void Unlock();
    CacheChain *LatchChain() const;
    void force( PageInfo *info );
    void force( PageInfo *info, a_page_count index );
```

```
  void finish() {
releaseRefs();
_is.deleted_at = FALSE;
_is.inserted_at = FALSE;
  }
  a_bool is_deleted() const {
return( _is.deleted_at );
  }
  a_bool is_on_page() const {
return( _name.as_member.page_no != NULL_PAGE );
  }
  void *page() const {
return( _info? _info->_image : NULL );
  }
  PageInfo *info() const {
return( _info );
  }
  a_page_id page_no() const {
return( _name.as_member.page_no );
  }
  a_page_count index() const {
return( _index );
  }
  a_bool valid() const {
return( !_is.page_deleted );
  }
  a_bool lookup_invalid() const {
return( _is.deleted_at || _is.inserted_at );
  }
  void read_lock_page() {
Lock( TRUE );
  }
  void read_lock_page( a_database_number db_no, a_page_id page_no ) {
Lock( NameFor( db_no, page_no ), TRUE );
  }
  void read_lock_couple( a_page_id page_no ) {
lock_couple( page_no, TRUE );
  }
  void write_lock_page() {
Lock();
  }
  void write_lock_page( a_database_number db_no, a_page_id page_no ) {
Lock( NameFor( db_no, page_no ) );
  }
  void write_lock_couple( a_page_id page_no ) {
lock_couple( page_no, FALSE );
  }
  void unlock_page();
  a_bool move_to( a_page_id page_no, a_page_count index );
  void lock( a_bool shared ) {
```

```
 Latch( shared );
 mark_dirty( shared );
   }
   void lock_couple( a_page_id page_no, a_bool shared );
   PageInfo *transfer_lock() {
// FIXME:  This is a hack that should be done away with.
_assertD( _is.locked );
_is.locked = FALSE;
return( _info->_image ? _info : NULL );
   }
   a_bool return_lock( CacheInfo * info ) {
_assertD( !_is.locked && info != NULL );
if( (PageInfo *) info == _info ) {
    _is.locked = TRUE;
    return( TRUE );
}
return( FALSE );
   }
   void releaseRefs();
   void lock_clean( a_bool shared ) {
Latch( shared );
   }
   void mark_dirty( a_bool shared = FALSE ) {
if( !shared && _info != NULL ) {
    _info->mark_dirty();
}
   }
   void write_lock_clean_page() {
Latch();
   }
   void unlock() {
_is.locked = FALSE;
_info->Unlock();
   }
protected:
   friend class Cache;
   void remove_from_locked_page();
   void clone( CacheRef const &ref );
   void latch() const;
public:
   CacheRef   *_next;
   CacheChain   *_chain;
   PageInfo   *_info;
   a_cache_name _name;
public:
   a_page_count _index;
   union {
struct {
    uint16 where : 8; // Where could we be in the scan?
    // LESS => value is less than left fencepost
```

```
    // EQUAL => value is between fenceposts
    // GREATER => value is greater than right fencepost
    // NB:
    // 1) Missing fenceposts essentially treated as infinite.
    // 2) After a comparison, only set according to value.
    uint16 pinned_to_page : 1;
    uint16 page_deleted : 1;
    uint16 reversed : 1;
    uint16 via_bitmap : 1;
    uint16 deleted_at : 1;
    uint16 inserted_at : 1;
    uint16 locked : 1;
 }  _is;
 uint16  _flags;
   };
protected:
   void latch_and_do_add_to_page( a_database_number db_no, a_page_id page_no );
};
class DBSRVAPI IOCB : public DeferredIO {
public:
   IOCB()
 : _info( NULL )
   {
   }
   void Latch() {
 _mutex.get();
   }
   a_bool TryLatch() {
 return( _mutex.try_get() );
   }
   void Unlatch() {
 _mutex.give();
   }
   a_bool Finish( a_bool wait, a_bool respect_evict );
   a_bool Finish( PageInfo * info, a_bool wait, a_bool respect_evict );
   a_bool DoFinish( PageInfo * info, a_bool wait, a_bool respect_evict );
   a_bool IsActuallyPending( void );
private:
   friend class DatabaseFile;
   friend class SysFile;
   friend class Cache;
   friend class PageInfo;
   PageInfo   *_info;
   an_ioreq_type _io_type; // PJB TODO: set counter type
   union {
 struct {
    uint8 to_be_evicted;
 }  _is;
 uint8  _flags;
   };
```

```
    uint32  _mask;
    Mutex  _mutex;
    CondVar  _finished;
};
#endif
// cachespt.cpp
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// ********************************************************************
// Copyright 1997-2003 iAnywhere Solutions, Inc.  All rights reserved.
// ********************************************************************
//                          -*- Mode: C++ -*-
//
// One stop shopping for memory.
//
// Contiguous array of infos point to exactly one image (that may or may
// not be addressable).  Address space is allocated immediately, and committed
// as needed.
// Age scores of pages as we check.  Have rover that checks.
//
// Replace grab_frame with explicit copy (locking pages).
// Use TryLock to reduce the number of times we need to give up a lock
// in order to delete a page.
//
// Have 2 reusable arrays: with and without address space.  Try for page
// with address space first, then for page without and troll for address
// space.
// Need quarantine (even for blocking hash implementation).  We can't reuse
// an info while an attempt is being made to latch it (but we can remove it).
// We put removed entries into a quarantine, and from there into the free
// queues.  We need per task hazard pointers.
//
// We should do a TryLatch operation first and only invoke the expensive
// operations when contention is detected.
//
// We could mark page as being quarantined (not eligible for reuse) if we
// need to block.  Have bits to mark info as free/guarded.  Mark as guarded
// if we need to block for latch.
// For old databases, keep page number of pre-image.  When attempting to
// to write a page with a pre-image, see if we can latch pre-image, and if
// so wait for IO to finish (will disappear from cache when done).
// Would like to wait for an iocb only if it belongs to a particular info
// We need a count of references + indication that would like to change name
// When count is 0 then we can change name; reset counter when we allocate
//
// ? Attempt to flush dirty page from cache -> start write, page marked
// as going out, unlock page -> attempt to read page -> get shared access
// and claim IOCB -> if we need page, do not evict.
//
// Resource orderings:
// Pages must be latched before pending IOCBs
```

```
// A page must be latched before its pre-image
#include "dbprecom.h"
#if defined( _MSC_VER )
   #pragma hdrstop
#endif
#include "cachespt.hpp"
#include "cachedef.hpp"
#include "rnthread.h" // for MAX_STACK_SIZE
#include "rnsqlpro.h"
#include "vmem.hpp"
#include "cachesize.hpp"
#include "dosbox.h"
#include "dbglobal.h"
#include "ksynch.hpp"
#include "dbstart.h"
#include "dbmulti.h"
#include "wcsupprt.hpp"
#include "pagemap.h"
#include "pagectr.hpp"
#include "rnconnec.h"
#include "ichkptlog.hpp"
#include "osintf.hpp"
#include "dbstate.h"
#include "dfsys.h"
#include "dfcio.h"
#include "dbyield.h"
#include "pagegrp.hpp"
#include "utlangstring.hpp"
#include "dbautockpt.hpp"
#include "storeifs.hpp"
#include "strids.h"
#include "cache.hpp"
#include "cachersv.hpp"
#include "cachedyn.hpp"
#include "cacheawe.hpp"
#include "dbcoldstart.hpp"
#include "dbtrace.hpp"
// #undef AWE_CACHE
// PJB FIXME:  To be removed.
inline p_Database LookupDBID( uint32 id ) {
   return( Eng->db.lookup( id ) );
}
inline p_Database DBFromID( uint32 id ) {
   return( LookupDBID( id ) );
}
#define _grant_image_access()
#define _revoke_image_access()
#if defined( VM_PROTECT )
   #define MIN_CACHE_PAGE_BITS 12
#elif defined( POINTERS_ARE_64BITS )
```

```
    // dec unix (which is a 64 bit platform) requires min of 2K page
    // this is due to some data structures which have exceeded the 1K page
    // limit due to the doubling of pointers in the structures.
    #define MIN_CACHE_PAGE_BITS 11
#else
    // all other platforms require at least 10 bits (1024 bytes)
    #define MIN_CACHE_PAGE_BITS 10
#endif
// #define MAX_IOCB 5 // make it small to test blocking logic
#if defined( UNDER_CE )
#define MAX_IOCB 50
#else
#define MAX_IOCB 255
#endif
#define ALL_FILES -1
#define MAPPED_PAGES -2
#if PRODUCTION < 1
a_bool DisableAssertNotFree = FALSE;
#endif
class DBSRVAPI IdleIOTimer : public ATimer {
public:
    IdleIOTimer();
    void dispatch();
private:
    unsigned _rover;
};
static Cache * XM = NULL;
CacheInterface * CM = NULL;
// PJB FIXME:  needs to be native.
typedef atomic32 atomic_ptrint;
class PageQueue {
public:
    PageQueue();
    void Initialize( PageInfo **queue );
    void Resize( Cache * cm, a_ptrint size );
    void Enqueue( PageInfo * info );
    PageInfo * Dequeue();
private:
    PageInfo **  _queue;
    a_ptrint  _size;
    a_ptrint  _threshold;
    atomic_ptrint _head;
    atomic_ptrint _tail;
};
class Cache : public CacheInterface {
public:
    Cache( p_engine_parms ep, AWEInterface * awe );
    ~Cache();
    a_bool IsUpAndRunning() const {
 return( XM == this );
```

```
    }
    an_image_set * get_images() {
  return( &_images );
    }
    PageInfo * LockRaw( a_cache_name name, a_bool shared = FALSE ) {
  PageInfo * info = Install( name, shared );
  info->FinishLock( shared );
  return( info );
    }
    PageInfo * Lock( a_cache_name name, a_bool shared = FALSE );
    unsigned Hint( a_cache_name name );
    void *AllocImage( CacheInfo ** pinfo );
    void FreeImage( CacheInfo * info );
    void *multi_page_alloc( uint32 num_pages, uint32 page_size, MultiPageAlloc *handle );
    void multi_page_free( MultiPageAlloc *handle );
    void Evict( a_cache_name name );
    void FlushDBFromCache( a_database_number db_no, a_bool is_scrammed ) {
  Evict( db_no, ALL_FILES, is_scrammed );
    }
    void FlushMappedPages( a_database_number db_no ) {
  Evict( db_no, MAPPED_PAGES, TRUE );
    }
    void FlushCacheForFileDrop( a_database_number db_no, unsigned file_no );
    void sa_flush_cache( p_database db );
    void Flush( p_database db );
    a_ptrint likely_available();
    a_ptrint __virtual__CacheMax() {
  // Outside of the cache manager, "CacheMax" is used as an
  // indication of the number of pages in the cache; however,
  // with dynamic cache sizing, _config._current.cache_max()
  // indicates the number of *frames* which could be larger
  // than the number of committed images if there are holes in
  // the image address space
  return( _config._current._n_committed_images );
    }
#if !PRODUCTION
    void CheckForLockedPages( int db_no );
#endif
    void adjust( DoAdjust *cb, Database *db, a_page_id p0, a_page_id p1, a_page_id p2 );
    a_bool WaitAny();
    void WaitAll( a_database_number db_no );
    void WaitForExtend( PageInfo * extend );
    void WaitUntilClean( IDatabaseFile * file );
    IOCB * GetIOCB( PageInfo * info );
    void PreIO();
    void PostIO();
    void Add( CacheRef * ref );
    // Depricated wrappers.
    void evict( Database *db, a_page_id page_no ) {
  Evict( NameFor( db->id(), page_no ) );
```

```
    }
    unsigned hint( Database * db, a_page_id page_no ) {
  return( Hint( NameFor( db->id(), page_no ) ) );
    }
    unsigned hint( a_page_id page_no ) {
  return( hint( _CurrentDB, page_no ) );
    }
    // NB Don't need DIO_Wait_All before call to this.
    void flush_db_from_cache( p_database db, a_bool is_scrammed ) {
  FlushDBFromCache( db->id(), is_scrammed );
    }
    void flush_mapped_pages( p_database db ) {
  FlushMappedPages( db->id() );
    }
    void flush_cache_for_table_drop( p_database db, unsigned file_no ) {
  FlushCacheForFileDrop( db->id(), file_no );
    }
    void flush( p_database db ) {
  Flush( db );
    }
    a_ptrint __virtual__cache_max()
    {
  return( __virtual__CacheMax() );
    }
#if !PRODUCTION
    void check_if_still_used_at_commit( p_database db, CacheInfo *def_page ) {
  CheckIfStillUsedAtCommit( db->id(), def_page );
    }
    void check_for_locked_pages( int db_no ) {
  CheckForLockedPages( db_no );
    }
#endif
    void hint_page_group( PageGroup *pagegrp, a_page_id start, uint32 pages_wanted );
    void cache_message( void );
    void engine_startup_complete();
    a_ptrint cache_available_images();
    a_ptrint cache_available_addr();
    a_bool enough_pages_available( uint32 num_pages, a_bool grow_cache_if_necessary );
    void kick_auto_resize();
    DynamicCacheConfig * min_config() {
  return( &_config._minimum );
    }
    DynamicCacheConfig * max_config() {
  return( &_config._maximum );
    }
    DynamicCacheConfig * current_config() {
  return( &_config._current );
    }
    DynamicCacheConfig * initial_config() {
  return( &_config._initial );
```

```
    }
    a_bool resize_cache( uint64 newsize );
#if defined(DYNAMIC_CACHE_SIZE)
    a_bool suspend_resizing();
    a_bool resume_resizing();
    a_bool resizing_suspended() const;
#endif
    uint64 current_cache_size() {
 return( _config._current.cache_size() );
    }
    uint64 minimum_cache_size() {
 return( _config._minimum.cache_size() );
    }
    uint64 maximum_cache_size() {
 return( _config._maximum.cache_size() );
    }
    CacheInfo *read_lock( p_database db, a_page_id page_no ) {
 return( Lock( NameFor( db->id(), page_no ), TRUE ) );
    }
    CacheInfo *write_lock( p_database db, a_page_id page_no ) {
 PageInfo * info = Lock( NameFor( db->id(), page_no ) );
 info->mark_dirty();
 return( info );
    }
    CacheInfo *write_lock_clean( p_database db, a_page_id page_no ) {
 return( Lock( NameFor( db->id(), page_no ) ) );
    }
    CacheInfo *write_lock_raw( p_database db, a_page_id page_no ) {
 return( LockRaw( NameFor( db->id(), page_no ) ) );
    }
    a_ptrint Clean( Database *db, unsigned rover, unsigned count );
private:
    friend class IOCB;
    friend class CacheRef;
    friend class PageQueue;
    a_ptrint ordinal( PageInfo * info ) {
 return( info - _info );
    }
    a_byte * ordinal_to_image( a_ptrint ordinal ) {
 return( _images.ordinal_to_image( ordinal ) );
    }
    CacheChain *ChainFor( a_cache_name name ) {
 return( &_page_hash[name.as_member.page_no%_hash_max] );
    }
    // Basic cache operations.
    PageInfo * AddToHash( CacheChain * chain, a_cache_name name, PageInfo * alloc );
    PageInfo * Alloc();
    void Free( PageInfo * info );
    void AddToReusable( PageInfo * info );
    PageInfo * AllocReusable();
```

```
    PageInfo * Install( a_cache_name name, a_bool shared = FALSE );
    CacheInfo * Install( a_database_file * f, a_page_id page_no, a_page_id real_page_no );
    PageInfo * IsNotInCache( a_cache_name name );
    PageInfo * IsInCache( a_cache_name name );
    PageInfo * LatchIfInCache( a_cache_name name );
    a_bool     IsImmediatelyLatchable( a_cache_name name );
    a_bool     IsIOPending( a_cache_name name );
    PageInfo * Scavenge();
    a_bool CanScavenge( PageInfo * info, class ScavengeState * state );
    PageInfo * Panic();
#if defined( AWE_CACHE )
#define TAG_UNMAPPED(x) ((void*)(((a_ptrint)(x))|1))
#define STRIP_TAG(x) ((a_byte*)(((a_ptrint)(x))&~1))
#define IS_UNMAPPED(x) ((((a_ptrint)(x))&1) != 0)
    void MakeAddressable( PageInfo * info ) {
 if( !info->IsAddressable() || IS_UNMAPPED( info->_image ) ) {
     DoMakeAddressable( info );
 }
    }
    void DoMakeAddressable( PageInfo * info );
#else
    void MakeAddressable( PageInfo * info ) {
 _unused( info );
    }
#endif
    a_bool RemoveFromHash( PageInfo * info );
    void Evict( a_database_number db_no, unsigned file_no, a_bool is_scrammed );
    void FlushPages( Database *db, struct a_flushed *flushed, unsigned n );
    void DoFlush( p_database db, a_bool flush_temp );
#if !PRODUCTION
    void CacheError( PageInfo *info, a_cache_error error );
    void CheckIfStillUsedAtFileDrop( a_database_number db_no, unsigned file_no );
    void CheckIfStillUsedAtCommit( a_database_number db_no, CacheInfo *def_page );
#endif
private:
    class IOGlobals {
    public:
 IOGlobals();
    private:
 friend class Cache;
 IOCB  iocb[MAX_IOCB];
 unsigned rover;
 IdleIOTimer idle_timer;
    }  _io;
private:
    friend class PageInfo;
    a_ptrint avail_infos() const {
 return( _config._current._n_infos );
    }
    // Page access.
```

```
    PageInfo *  _info;  // Contiguous array of all infos.
    unsigned  _rover;  // cycles through infos
    PageQueue  _reusable; // a reusable info
    atomic_ptrint _pinned_images;
    // Page naming.
    CacheChain * _page_hash; // Where the pages live.
    unsigned  _hash_max; // Number of chains.
    // Page scoring.
#define SEGMENTS_IN_CACHE 32
#define MAX_SCORE  SEGMENTS_IN_CACHE
#define MAX_THRESHOLD  (SEGMENTS_IN_CACHE/2)
    unsigned  _now;  // Current "time".
    unsigned  _delta;  // Window size (increment score).
    unsigned  _incr;
    void SetPageScoringParameters() {
 _delta = cache_available_images()/SEGMENTS_IN_CACHE;
 if( _delta == 0 ) _delta = 1;
 _incr = sqrt((double)_delta);
    }
private:
    void assert_page_no( Database *db, a_page_id page_no, void *page ) {
 // This check is hard-wired for performance reasons
 if( *(uint32 *)page != page_no ) {
     dump_page( db, page, page_no );
     _assertP( 101412, FALSE,
        "Page number on page does not match page requested" );
 }
    }
    an_image_set _images;
#if defined( AWE_CACHE )
private:
    AWECache  _awe_cache;
    PageInfo **  _frame_owner;
    a_ptrint  _frame_rover;
#endif
    a_bool awe_enabled() {
    // The code is cleaner in some places if this function is defined
    // on non-awe platforms as well.
#if defined( AWE_CACHE )
 return( _awe_cache.enabled() );
#else
 return( FALSE );
#endif
    }
    CacheAllocator _allocator;
    void recompute_headroom() {
#if defined(DYNAMIC_CACHE_SIZE)
 _headroom = _auto_resize? _config._maximum._n_committed_images -
     _config._current._n_committed_images : 0;
#else
```

```
 _headroom = 0;
#endif
    }
    a_ptrint  _headroom;
private:
    a_bool GrowInfos( a_ptrint to_commit );
    a_ptrint CommitImages( a_ptrint lo, a_ptrint to_commit );
    a_bool recommit( void *p, a_ptrint old_count, a_ptrint new_count, a_ptrint item_size );
#if defined(DYNAMIC_CACHE_SIZE)
private:
    void after_cache_size_change();
    a_ptrint GrowImages( a_ptrint to_commit );
    a_bool grow( a_ptrint to_commit );
    a_bool grow_cache( DynamicCacheConfig &config );
    a_bool grow_cache_by_images( a_ptrint num_images );
#if !PRODUCTION
    void debug_images();
#else
    void debug_images() {}
#endif
#if defined( ALLOW_CACHE_TO_SHRINK )
    a_bool IsShrinkable( PageInfo * info );
#endif
    void shrink_images( a_ptrint to_decommit );
    a_ptrint FindRange( a_ptrint hi, a_bool is_shrinkable );
    a_ptrint Decommit( a_ptrint lo, a_ptrint hi, a_ptrint max );
    a_bool shrink_cache( DynamicCacheConfig &config );
private:
    Mutex  _cache_growth_mutex;
    CacheAutoResizeTimer *_auto_resize;
#endif
    CacheBounds  _config;
#if defined( LINUX )
     a_bool  _touch_page_before_use;
#endif
};
static inline void
SuicideOnFatalError()
/*******************/
{
    if( DB_Fatal_error != SQLSTATE_NOERROR ) {
 _CurrentWorker->suicide( -1 );
    }
}
CacheInterface::CacheInterface( p_engine_parms ep )
/************************************************/
    : _db_page_bits_max ( ep->page_bits     )
    , _db_page_size_max ( 1<<_db_page_bits_max    )
    , _db_page_mask_max ( - (a_ptrint)_db_page_size_max   )
    , _os_page_size ( OS_PageSize()     )
```

```
    , _os_page_bits ( log2( _os_page_size )    )
    , _alignment_amount ( _max( _db_page_size_max, _os_page_size ) )
    , _ordinal_alignment( _alignment_amount/_db_page_size_max  )
{
    CM = this;
}
CacheInterface::~CacheInterface()
/*******************************/
{
}
PageQueue::PageQueue()
/*********************/
    : _queue( NULL )
    , _size ( 0 )
    , _threshold( 3 ) // PJB FIXME: should be based on number of threads
    , _head ( 0 )
    , _tail (0 )
{
}
void
PageQueue::Initialize( PageInfo **queue )
/***************************************/
{
    _queue = queue;
}
void
PageQueue::Resize( Cache * cm, a_ptrint size )
/********************************************/
{
    if( cm->recommit( _queue, _size, size, sizeof(PageInfo *) ) ) {
 _size = size;
 PageInfo * high = &cm->_info[cm->_config._current._n_infos];
 for( a_ptrint i = 0; i < size; ++i ) {
     if( _queue[i] >= high ) {
  _queue[i] = NULL;
     }
 }
    }
}
void
PageQueue::Enqueue( PageInfo * info )
/***********************************/
{
    if( ((a_ptrint) (info->_queue_slot - _tail)) >= _size ||
     _queue[info->_queue_slot%_size] != info ) {
 if( (_head - _tail) < (_size - _threshold) ) {
     _queue[(info->_queue_slot = _head++)%_size] = info;
 }
    }
}
```

```
PageInfo *
PageQueue::Dequeue()
/*****************/
{
    return( (_head - _tail) > _threshold? _queue[(_tail++)%_size] : NULL );
}
static void
Check_available_physical_memory( uint64 cache_size, uint64 phys_avail )
/*********************************************************************/
{
#if !defined( WINNT ) && !defined( UNIX )
    _unused( cache_size );
    _unused( phys_avail );
#else
    #if defined( WINNT )
 if( IsWindows95() ) return;
    #endif
    #if defined( UNIX )
        if( OS_TotalPhysicalMemory() == 0 ) return;
    #endif
    if( cache_size > phys_avail ) {
 startup_msg( IDS_ENG_CACHE_EXCEEDS_PHYS_MEM,
       (uint32)(cache_size/_u64_const(1024)),
       (uint32)(phys_avail/_u64_const(1024)) );
    }
#endif
}
Cache::Cache( p_engine_parms ep, AWEInterface * awe )
/***************************************************/
    : CacheInterface( ep )
#if defined( AWE_CACHE )
    , _awe_cache( this, awe )
#endif
    , _images( _db_page_bits_max )
    , _now( 0 )
    , _rover( 0 )
{
    uint64 phys_avail;
#if !defined( AWE_CACHE )
    _unused( awe );
#endif
    _pinned_images = 0;
    // Save the amount of physical memory available at this point in time. On platforms
    // where we cannot reserve/commit memory separately, we will want to compare the cache
    // size against this value later.
    phys_avail = OS_AvailablePhysicalMemory();
    // Determine possible range of cache sizes.
    _config.init( ep, awe_enabled() );
    if( !awe_enabled() ) {
 if( !_allocator.init( maximum_cache_size(), _alignment_amount ) ) {
```

```
    return;
  }
  // Cache size may have changed.
  if( !_config.fixup( _allocator.size() ) ) {
    return;
  }
   }
#if defined( AWE_CACHE )
   else {
  if( !_allocator.init( _config._maximum.cache_size_without_images(), _alignment_amount ) ) {
    return;
  }
   }
#endif
   _config._initial = _config._current; // save for future reference
   if( !awe_enabled() ) {
  Check_available_physical_memory( _config._initial.cache_size(), phys_avail );
   }
   #if defined( DYNAMIC_CACHE_SIZE )
  _auto_resize = ep->auto_resize? new CacheAutoResizeTimer( this ) : NULL;
   #endif
   recompute_headroom();
   #if defined( DYNAMIC_CACHE_SIZE )
  #define _pick_alignment(static_cache,dynamic_cache) (dynamic_cache)
   #else
  #define _pick_alignment(static_cache,dynamic_cache) (static_cache)
   #endif
   a_ptrint to_commit = _config._current._n_committed_images;
   _config._current._n_infos = 0;
   void * alloc =
  _allocator.alloc( 0, _config._maximum._n_infos, sizeof(PageInfo),
    _pick_alignment( sizeof(uint64), _os_page_size ) );
   if( alloc == NULL ) return;
   _info = (PageInfo *) alloc;
   alloc =
  _allocator.alloc( 0, _config._maximum._n_reusable, sizeof(PageInfo*),
    _pick_alignment( sizeof(uint64), _os_page_size ) );
   if( alloc == NULL ) return;
   _reusable.Initialize( (PageInfo **) alloc );
   _reusable.Resize( this, _config._current._n_reusable );
   // FIXME: _hash_max does not currently change when the cache changes size.
   _hash_max = _config._current._n_hash_chains;
   _page_hash = (CacheChain *)
  _allocator.alloc( _hash_max,
    _config._maximum._n_hash_chains,
    sizeof(CacheChain),
    _pick_alignment( sizeof(uint64), _os_page_size ) );
   if( _page_hash == NULL ) return;
   for( unsigned i = 0; i < _hash_max; ++i ) {
      new ( &_page_hash[i] ) CacheChain;
```

```
  }
  if( !awe_enabled() ) {
_assertD( to_commit == _config._current._n_committed_images );
_config._current._n_committed_images = 0;
unsigned num_pieces;
const an_aligned_allocation_piece *p =
   _allocator.alloc_piecewise( 0, _config._maximum._n_committed_images,
   _db_page_size_max,
   _pick_alignment( _db_page_size_max,
     _alignment_amount ),
   MAX_IMAGE_ADDRESS_SPACE_RANGES,
   &num_pieces );
if( p == NULL ) return;
for( unsigned i = num_pieces; i-- > 0; ) {
   _images.set_range( i, p[i].mem(), p[i].size() );
}
// PJB FIXME:  Too drastic?
if( !GrowInfos( to_commit ) ) return;
if( CommitImages( 0, to_commit ) != 0 ) return;
  }
  #if defined( AWE_CACHE )
  else {
if( !_awe_cache.allocate_physical_memory( ep, _config._current.size_of_images() ) ) {
   return;
}
_frame_owner = (PageInfo **) malloc(_awe_cache._addr_space*sizeof(PageInfo*));
if( _frame_owner == NULL ) {
   return;
}
memset( _frame_owner, 0, _awe_cache._addr_space*sizeof(PageInfo*));
_frame_rover = 0;
if( _awe_cache._n_dbpage_frames < to_commit ) {
   startup_msg( IDS_ENG_AWE_DID_NOT_ALLOCATE_EXPECTED_PHYSMEM,
  (uint32)(((uint64)to_commit*(uint64)_db_page_size_max) / _u64_const(1024)),
  (uint32)(((uint64)_awe_cache._n_dbpage_frames*(uint64)_db_page_size_max) / _u64_const(1024)) );
   to_commit = _awe_cache._n_dbpage_frames;
   _config._current._n_committed_images = _awe_cache._n_dbpage_frames;
}
if( !_awe_cache.allocate_address_space() ) {
   return;
}
if( !GrowInfos( to_commit ) ) return;
// Images will gain addressability on demand.
for( unsigned i = 0; i < _config._current._n_infos; ++i ) {
   PageInfo * info = &_info[i];
   info->_image = NULL;
   info->Committed();
}
  }
  #endif
```

```
    SetPageScoringParameters();
    XM = this;
    _engine_statistic_set( CACHE_PINNED, _pinned_images );
    _engine_statistic_set( MAIN_HEAP_PAGES, 0 );
    CacheInfo *  info;
    void  *image = AllocImage( &info );
    DV_InitMainHeap( image, info, _db_page_size_max );
    _engine_statistic_set( CURRENT_CACHE_SIZE, current_cache_size()/1024 );
    _engine_statistic_set( PEAK_CACHE_SIZE,    current_cache_size()/1024 );
    _engine_statistic_set( MIN_CACHE_SIZE,     minimum_cache_size()/1024 );
    _engine_statistic_set( MAX_CACHE_SIZE,     maximum_cache_size()/1024 );
    #if defined( LINUX )
  _touch_page_before_use = FALSE;
  int major, minor, patch;
  if( OS_Version( &major, &minor, &patch ) > 0 ) {
      if( major == 2 && minor == 2 && patch < 17 ) {
   _touch_page_before_use = TRUE;
      }
  }
    #endif
}
a_ptrint
Cache::CommitImages( a_ptrint lo, a_ptrint to_commit )
/*****************************************************/
{
    a_ptrint hi = _config._current._n_infos;
    for( ; lo < hi && to_commit > 0; ++lo ) {
  a_ptrint tagged = 0;
  for(; tagged < to_commit && lo + tagged < hi && _info[lo+tagged].CanCommit(); ++tagged );
  if( tagged > 0 ) {
      a_ptrint aligned_lo = round_up_ordinal( lo );
      a_ptrint aligned_hi = round_down_ordinal( lo + tagged );
      if( aligned_lo < aligned_hi ) {
   a_ptrint count = _images.commit( aligned_lo, aligned_hi - aligned_lo );
   for( a_ptrint i = 0; i < count; ++i ) {
       _info[aligned_lo+i].Committed();
   }
   _config._current._n_committed_images += count;
   recompute_headroom();
   to_commit -= count;
      }
      lo += tagged;
  }
    }
    return( to_commit );
}
a_bool
Cache::GrowInfos( a_ptrint to_commit )
/************************************/
{
```

```
    a_ptrint original = _config._current._n_infos;
    a_ptrint requested = original + to_commit;
    if( recommit( _info, original, requested, sizeof(PageInfo) ) ) {
 for( a_ptrint ordinal = original; ordinal < requested; ++ordinal ) {
     new (&_info[ordinal]) PageInfo( ordinal_to_image( ordinal ) );
 }
 _config._current._n_infos = requested;
 return( TRUE );
    }
    return( FALSE );
}
a_bool
Cache::recommit( void *p, a_ptrint old_count, a_ptrint new_count, a_ptrint item_size )
/*********************************************************************************************/
{
#if defined( DYNAMIC_CACHE_SIZE )
    _assertD( (((a_ptrint)p)&(_os_page_size-1)) == 0 );
    _assertD( item_size <= _os_page_size );
    a_ptrint old_size = (a_ptrint) _round_up_pow2( old_count * item_size, _os_page_size );
    a_ptrint new_size = (a_ptrint) _round_up_pow2( new_count * item_size, _os_page_size );
    if( new_size > old_size ) {
 return( Vmem_commit( ((char *)p)+old_size, new_size-old_size ) );
    } else if( new_size < old_size ) {
 return( Vmem_decommit( ((char *)p)+new_size, old_size-new_size ) );
    }
#else
    _unused( p );
    _unused( old_count );
    _unused( new_count );
    _unused( item_size );
#endif
    return( TRUE );
}
// PJB FIXME:  Next two routines not quite right.
a_bool
PageInfo::CanCommit()
/*******************/
{
    return( IsMissingImage() );
}
void
PageInfo::Committed()
/*******************/
{
    MarkImageAsAvailable();
}
#if defined( DYNAMIC_CACHE_SIZE )
void
Cache::after_cache_size_change()
/******************************/
```

```
{
    SetPageScoringParameters();
    a_ptrint size_in_k = current_cache_size()/1024;
    _engine_statistic_set( CURRENT_CACHE_SIZE, size_in_k );
    if( size_in_k > *_engine_counter( PEAK_CACHE_SIZE ) ) {
 _engine_statistic_set( PEAK_CACHE_SIZE, size_in_k );
    }
    DB_Message( IDS_ENG_CACHE_SIZE_CHANGED, size_in_k );
    #if !PRODUCTION
        LSBuf msg;
 #if defined( ALLOW_CACHE_TO_SHRINK )
     a_ptrint min_infos = DynamicCacheConfig::minimum_n_infos( _config._current._n_committed_images );
     if( _config._current._n_infos > min_infos ) {
  sprintf( msg, "Cache config: %ld images, %ld[%ld ideal] infos",
    _config._current._n_committed_images, _config._current._n_infos, min_infos );
  DB_Message( msg );
  return;
     } else
        #endif
     {
  sprintf( msg, "Cache config: %ld images, %ld infos",
    _config._current._n_committed_images, _config._current._n_infos );
  DB_Message( msg );
     }
    #endif
}
#if defined( ALLOW_CACHE_TO_SHRINK )
a_bool
PageInfo::IsShrinkable()
/**********************/
{
    return( _pending == NULL && !_is.dirty && !IsLatched() );
}
a_bool
Cache::IsShrinkable( PageInfo * info )
/************************************/
{
    if( !info->IsShrinkable() ) {
 return( FALSE );
    }
    if( info->IsInHash() ) {
 a_cache_name name = info->name();
 CacheChain * chain = ChainFor( name );
 for( CacheRef * ref = chain->_refs; ref != NULL; ref = ref->_next ) {
     if( ref->_name.as_uint == name.as_uint ) {
  return( FALSE );
     }
 }
    }
    return( TRUE );
```

```
}
#if !PRODUCTION
unsigned fake_out_opt;
void
Cache::debug_images()
/*******************/
{
    for( a_ptrint i = 0; i < avail_infos(); ++i ) {
 if( !_info[i].IsMissingImage() ) {
     fake_out_opt += * (unsigned *) (_info[i]._image);
 }
    }
}
#endif
void
PageInfo::Shrink( Cache * cm )
/****************************/
{
    Latch();
    if( IsInHash() ) {
 cm->RemoveFromHash( this );
    }
    MarkImageAsMissing();
    _name.as_uint = BOGUS_NAME;
    Unlatch();
}
a_ptrint
Cache::Decommit( a_ptrint lo, a_ptrint hi, a_ptrint max )
/******************************************************/
{
    a_ptrint aligned_lo = round_up_ordinal( lo );
    a_ptrint aligned_hi = round_down_ordinal( hi );
    if( aligned_hi < aligned_lo ) aligned_hi = aligned_lo;
    a_ptrint count = aligned_hi - aligned_lo;
    if( count > max ) count = max;
    if( count > 0 ) {
 debug_images();
 aligned_lo = aligned_hi - count;
 _images.decommit( aligned_lo, count );
 _config._current._n_committed_images -= count;
 recompute_headroom();
 for( a_ptrint i = 0; i < count; ++i ) {
     _info[aligned_lo + i].Shrink( this );
 }
 if( aligned_hi == _config._current._n_infos ) {
     _config._current._n_infos = aligned_lo;
     for( a_ptrint i = 0; i < count; ++i ) {
  _info[aligned_lo + i].~PageInfo();
     }
     recommit( _info, aligned_hi, aligned_lo, sizeof(PageInfo) );
```

```
 }
 debug_images();
    }
    return( count );
}
a_ptrint
Cache::FindRange( a_ptrint hi, a_bool is_shrinkable )
/************************************************/
{
    while( hi-- > 0 ) {
 if( IsShrinkable( &_info[hi] ) != is_shrinkable ) break;
    }
    return( hi + 1 );
}
void
Cache::shrink_images( a_ptrint to_decommit )
/*****************************************/
{
    _assertD( !awe_enabled() );
    to_decommit = round_down_ordinal( to_decommit );
    a_ptrint lo = _config._current._n_infos;
    while( to_decommit > 0 && lo > 0 ) {
 a_ptrint hi = FindRange( lo, FALSE );
 lo = FindRange( hi, TRUE );
 to_decommit -= Decommit( lo, hi, to_decommit );
    }
}
a_bool
Cache::shrink_cache( DynamicCacheConfig &new_config )
/************************************************/
{
    _assertD( !awe_enabled() );
    a_ptrint committed = _config._current._n_committed_images;
    if( new_config._n_committed_images > committed ) {
 // Even though the cache size has decreased, it may cause the number of
 // images to increase due to other structures decreasing in size and
 // due to the way in which we compute the size of the various cache
 // components. All non-image components must shrink or remain the same
 // size.
 return( FALSE );
    }
    // It is safe to wait for all IOs to complete while in SuperForbid state
    // because no worker can be doing synchronous IO if we are SuperForbidding
    // and the current worker can handle the completion of all async IOs
    DIO_Wait_all();
    shrink_images( committed - new_config._n_committed_images );
    _reusable.Resize( this, new_config._n_reusable );
    _config._current._n_reusable = new_config._n_reusable;
    // PJB TODO:  fix check.
#if !PRODUCTION && 0
```

```
    CacheChain *chain;
    for( i=0; i<_hash_max; i++ ) {
  chain = &_page_hash[i];
  for( info=chain->_head; info; info=info->_next ) {
      _assertD( info < max_info );
  }
    }
#endif
    // update the queues with meaningful info
    after_cache_size_change();
    return( TRUE );
}
#endif // ALLOW_CACHE_TO_SHRINK
a_ptrint
Cache::GrowImages( a_ptrint to_commit )
/************************************/
{
    a_ptrint orig = _config._current._n_infos;
    if( GrowInfos( to_commit ) ) {
  to_commit = CommitImages( orig, to_commit );
    }
    return( to_commit );
}
a_bool
Cache::grow( a_ptrint to_commit )
/*******************************/
{
    a_ptrint requested = to_commit;
    to_commit = CommitImages( 0, to_commit );
    if( to_commit > 0 ) {
  to_commit = GrowImages( to_commit );
    }
    return( to_commit != requested );
}
a_bool
Cache::grow_cache( DynamicCacheConfig &new_config )
/*************************************************/
// Must be holding _cache_growth_mutex
{
    DynamicCacheConfig old_config;
    old_config = _config._current;
    if( new_config._n_committed_images < old_config._n_committed_images ) {
  // Even though the cache size has increased, it may cause the number of
  // images to decrease due to other structures increasing in size and
  // due to the way in which we compute the size of the various cache
  // components. All non-image components must grow or remain the same size.
  return( FALSE );
    }
    _reusable.Resize( this, new_config._n_reusable );
    _config._current._n_reusable = new_config._n_reusable;
```

```
    // If there are holes in the image array then _n_infos may need to
    // remain larger than number of images in the desired new configuration
    new_config._n_infos = _max( new_config._n_infos, old_config._n_infos );
    if( !grow( new_config._n_committed_images - old_config._n_committed_images ) ) {
 return( FALSE );
    }
    // update the queues with meaningful info
    after_cache_size_change();
    return( TRUE );
}
a_bool
Cache::grow_cache_by_images( a_ptrint images_to_add )
/*************************************************/
// Must already hold the _cache_growth_mutex
{
    // If we try to grow the cache by too small a value, the rounding
    // of the sizes of the various cache data structures may not
    // result in an increase in the number of images available
    if( awe_enabled() ) {
 return( FALSE );
    }
    #define MIN_IMAGES_TO_ADD 64
    images_to_add = _max( images_to_add, MIN_IMAGES_TO_ADD );
    images_to_add = round_up_ordinal( images_to_add );
    DynamicCacheConfig config = _config._current;
    config._n_committed_images += images_to_add;
    if( config._n_committed_images > config._n_infos ) {
 config._n_infos = config._n_committed_images;
    }
    config._n_reusable = config._n_infos;
    if( config.cache_size() > maximum_cache_size() ) {
 return( FALSE );
    }
    return( grow_cache( config ) );
}
a_bool
Cache::resize_cache( uint64 newsize )
/***********************************/
// Do the best we can to make the cache size approximately "newsize".
// Returns true if cache size changed
{
    a_bool  changed;
    DynamicCacheConfig config;
    if( awe_enabled() ) {
 return( FALSE );
    }
    newsize = _max( newsize, minimum_cache_size() );
    newsize = _min( newsize, maximum_cache_size() );
    config.recompute( newsize );
    if( config.cache_size() > maximum_cache_size() ||
```

```
      config.cache_size() < minimum_cache_size() ) {
 return( FALSE );
   }
   _cache_growth_mutex.get();
   if( config.cache_size() > current_cache_size() ) {
 // We cannot SuperForbid() when growing the cache because
 // we may be trying to grow the cache in a panic in which case
 // we may have pages locked that other tasks are waiting for
 changed = grow_cache( config );
 _cache_growth_mutex.give();
   } else if( config.cache_size() < current_cache_size() ) {
 _cache_growth_mutex.give();
 #ifdef ALLOW_CACHE_TO_SHRINK
    // Note: _config._current can change here but we cannot hold the
    // _cache_growth mutex when we call SuperForbid because one of the
    // workers may be trying to grow the cache (possibly in a panic)
    SuperForbid();
    // Note that nobody can be holding the _cache_growth_mutex at this point
    if( config.cache_size() < current_cache_size() ) {
  changed = shrink_cache( config );
    }
    SuperPermit();
 #else
    changed = FALSE;
 #endif
   } else {
 _cache_growth_mutex.give();
 changed = FALSE;
   }
   return( changed );
}
#else
a_bool
Cache::resize_cache( uint64 newsize )
/***********************************/
{
   _unused( newsize );
   return( FALSE );
}
#endif
a_ptrint
Cache::cache_available_images()
/*****************************/
{
   a_ptrint total = avail_infos();
   a_ptrint pinned = _pinned_images;
   return( total > pinned? total - pinned : 0 );
}
a_ptrint
Cache::cache_available_addr()
```

```
/**************************/
{
#if defined( AWE_CACHE )
    if( awe_enabled() ) {
 return( _pinned_images < _awe_cache._addr_space
  ?  (_awe_cache._addr_space - _pinned_images)*_db_page_size_max
  : 0 );
    }
#endif
    return( cache_available_images() );
}
a_bool
Cache::enough_pages_available( uint32 num_pages, a_bool grow_cache_if_necessary )
/*****************************************************************************/
// Assumes that the caller is looking to grow a large, locked heap so we care
// that enough *address* space is available for the number of pages
{
#if defined( DYNAMIC_CACHE_SIZE )
    if( cache_available_addr() >= num_pages ) {
 return( TRUE );
    }
    if( !grow_cache_if_necessary || awe_enabled() ) {
 return( FALSE );
    }
    _cache_growth_mutex.get();
    a_ptrint avail = cache_available_addr();
    if( avail >= num_pages ) {
 _cache_growth_mutex.give();
 return( TRUE );
    }
    grow_cache_by_images( num_pages - avail );
    a_bool enough = ( cache_available_addr() >= num_pages );
    _cache_growth_mutex.give();
    return( enough );
#else
    _unused( grow_cache_if_necessary );
    return( cache_available_addr() >= num_pages );
#endif
}
#if !PRODUCTION
// Use this routine like BMem in dballoc.c.
typedef struct mp_alloc {
    struct mp_alloc  *next;
    void   *contents;
    a_debug_count count;
} mp_alloc;
mp_alloc   *MultiPageAllocs = NULL;
atomic32  MultiPageCount = 0;
a_debug_count  WMPmem = 0;
static void BMPmem( a_debug_count watch )
```

```
/*************************************/
// Set WMPmem to the count you want, then set a breakpoint at BMPmem
{
    WMPmem = watch;       // prevent optimizer from merging functions
}
static void CheckMultiPageAllocs()
/*********************************/
// Verify that all multi-page allocations have been freed.
{
    _assertD( MultiPageAllocs == NULL );
}
#endif
Cache::~Cache()
/*************/
{
    unsigned i;
    _db_declare_yield( DBFINIHASH1, 30 );
#if defined( DYNAMIC_CACHE_SIZE )
    if( _auto_resize != NULL ) {
 delete _auto_resize;
    }
#endif
    DV_Fini();
    #if !PRODUCTION
 CheckMultiPageAllocs();
    #endif
    for( i=0; i < _hash_max; ++i ) {
 /* Must call destructor to destroy Mutex - in Netware, open semaphores
    are not closed by the OS, and if we don't do it, the OS crashes. */
        _page_hash[i].~CacheChain();
 _db_yield( DBFINIHASH1 );
    }
#if defined( AWE_CACHE )
    free( _frame_owner );
#endif
}
a_bool
CM_Init( p_engine_parms ep )
/**************************/
{
    if( ep->page_bits < MIN_CACHE_PAGE_BITS ) {
 ep->page_bits = MIN_CACHE_PAGE_BITS;
    }
    return( (new Cache( ep, AWECache::AllocInterface( ep ) ))->IsUpAndRunning() );
}
void
CM_Fini()
/*******/
{
    delete CM;
```

```
}
#if defined(DYNAMIC_CACHE_SIZE)
a_bool
Cache::suspend_resizing()
/**********************/
{
   if( _auto_resize != NULL ) {
      _auto_resize->suspend();
      return TRUE;
   } else {
      return FALSE;
   }
}
a_bool
Cache::resume_resizing()
/*********************/
{
   if( _auto_resize != NULL ) {
      _auto_resize->resume();
      return TRUE;
   } else {
      return FALSE;
   }
}
a_bool
Cache::resizing_suspended() const
/*******************************/
{
   if( _auto_resize != NULL ) {
      return _auto_resize->suspended();
   } else {
      return FALSE;
   }
}
#endif
a_bool CM_ResizeCache( uint64 newsize )
/************************************/
{
   a_bool ret = XM->resize_cache( newsize );
#if defined(DYNAMIC_CACHE_SIZE)
   XM->suspend_resizing();
#endif
   return ret;
}
a_bool CM_ResumeResizing()
/************************/
{
#if defined(DYNAMIC_CACHE_SIZE)
   return( XM->resume_resizing() );
#else
```

```
    return TRUE;
#endif
}
uint64 CM_CurrentCacheSize()
/*************************/
{
    return( XM->current_cache_size() );
}
uint64 CM_MinimumCacheSize()
/*************************/
{
    return( XM->minimum_cache_size() );
}
uint64 CM_MaximumCacheSize()
/*************************/
{
    return( XM->maximum_cache_size() );
}
a_bool
CM_EnoughPagesAvailable( uint32 num_pages, a_bool grow_cache_if_necessary )
/*****************************************************************************/
{
    return( XM->enough_pages_available( num_pages, grow_cache_if_necessary ) );
}
a_ptrint
Cache::likely_available()
/***********************/
// Assumes that the caller is looking to grow a large, locked heap so we care
// that enough *address* space is available for the number of pages
{
#if defined(DYNAMIC_CACHE_SIZE)
    if( resizing_suspended() ) {
        return( cache_available_addr() );
    } else {
        return( cache_available_addr() + _headroom );
    }
#else
    return( cache_available_addr() + _headroom );
#endif
}
a_ptrint
CM_LikelyAvailable()
/******************/
{
    return( XM->likely_available() );
}
#if !PRODUCTION
static void sem_dump( void )
/**************************/
// Dump latches to a file, hit the debugger.
```

```
{
    KSem_dump();
    _assertD( FALSE );
}
#if 0 // unused
static void DbgPageLock( void )
/*****************************/
{
#if 0
    // Keep track of number of page locks owned by current task.
    ++_taskData( page_locks );
#endif
}
static void DbgPageUnlock( PageInfo *info )
/*****************************************/
{
#if 0
    // Keep track of number of page locks owned by current task.
    //    _assertD( _taskData( page_locks ) != 0 );
    if( _taskData( page_locks ) == 0 ) {
 sem_dump();
    }
    _taskData( page_locks )--;
    // Do some page consistency checks on table pages if they have been
    // modified.
    info->_file->consistency_check( frame->_image, info->_owner != NULL );
#else
    _unused( info );
#endif
}
#endif // unused
void
CM_AssertNoPageLocks( void )
/**************************/
{
    if( _taskData( page_locks ) != 0 ) {
 sem_dump();
    }
}
#endif
#if !PRODUCTION
void
Cache::CacheError( PageInfo *info, a_cache_error error )
/******************************************************/
{
    char  *msg;
    switch( error ) {
    case CACHE_ERR_WRITE_LOCK:
 msg = "modified without write lock";
 break;
```

```
    case CACHE_ERR_UNLOCK:
  msg = "cannot be unlocked";
  break;
    case CACHE_ERR_STILL_LOCKED:
  msg = "left locked";
  break;
    default:
  msg = "Unknown error";
    }
#if !defined( UNDER_CE )
    // Cache page num is wrong for SEGMENTED_CACHE, but don't worry about it
    // PJB FIXME:  Add locking info back to msg
    char buf[256];
    sprintf( buf, "cache page %d (page %ld) %s\n",
     ordinal( info ),
     info->_file->get_page_no( info->_image ),
     msg );
    DB_Message( buf );
#else
    _unused( info );
    OutputDebugString( msg );
#endif
    _assertP( 101402, FALSE,
       "Cache error" );
}
void
Cache::CheckForLockedPages( int db_no )
/*************************************/
{
#if 1
    // PJB FIXME:  Implement.  Note that pages can be locked by other
    // threads attempting to claim page.
    _unused( db_no );
#else
    for( a_ptrint i = 0; i < avail_infos(); i++ ) {
  PageInfo * info = &_info[i];
  if( !_is_temporary( info->page_no() )
      && info->db_no() == db_no && info->IsLocked() ) {
      CacheError( info, CACHE_ERR_STILL_LOCKED );
  }
    }
#endif
}
void
Cache::CheckIfStillUsedAtFileDrop( a_database_number db_no, unsigned file_no )
/*****************************************************************************/
{
    // Check to see if any pages are still in use.
    for( a_ptrint i = 0; i < avail_infos(); i++ ) {
  PageInfo * info = &_info[i];
```

```
 if( info->db_no() == db_no
     && _file_num( info->page_no() ) == file_no ) {
     // PJB FIXME:  Wrong error constant.
     CacheError( info, CACHE_ERR_STILL_LOCKED );
 }
    }
}
#endif
////////////////////////////////////////////////////////////////////////////
void
CacheRef::Remove()
/****************/
{
    CacheChain * chain;
    while( (chain = _chain) != NULL && !chain->Remove( this ) );
}
void
CacheRef::Latch( a_bool shared )
/******************************/
{
    PageInfo * info;
    for( ;; ) {
 a_cache_name name = _name;
 // PJB FIXME:  abstract
 if( _chain == NULL ) return;
 info = _info;
 if( info == NULL || !info->Latch( XM, name, shared ) ) {
     if( _chain == NULL ) return;
     if( _name.as_uint != name.as_uint ) continue;
     info = XM->Install( name, shared );
 }
 info->FinishLock( shared );
 if( _name.as_uint == info->name().as_uint ) break;
 info->Unlock();
 _info = NULL;
    }
    _info = info;
    _assertD( !_is.locked );
    _is.locked = TRUE;
}
CacheChain *
CacheRef::LatchChain() const
/********************/
{
    CacheChain * chain;
    while( (chain = _chain) != NULL ) {
 chain->Latch();
 if( _chain == chain ) break;
 chain->Unlatch();
    }
```

```
    return( chain );
}
void
CacheRef::Unlock()
/****************/
{
    if( _is.locked ) {
 _is.locked = FALSE;
 _info->Unlock();
    }
}
void
CacheRef::releaseRefs()
/*********************/
{
    if( _chain != NULL ) {
 if( _is.locked ) {
     _chain->Remove( this );
     Unlock();
 } else {
     Remove();
 }
    }
    _assertD( !_is.locked );
    _info = NULL;
    _name.as_member.page_no = NULL_PAGE;
}
void
CacheRef::clone( CacheRef const & ref )
/************************************/
{
    CacheChain * chain = ref.LatchChain();
    if( chain != NULL ) {
 _chain = chain;
 _info = ref._info;
 _index = ref._index;
 _flags = ref._flags;
 _is.locked = FALSE;
 _name = ref._name;
 _next = chain->_refs;
 chain->_refs = this;
 chain->Unlatch();
    } else {
 _chain = NULL;
 _info = NULL;
 _name.as_member.page_no = NULL_PAGE;
 _index = ref._index;
 _flags = ref._flags;
 _is.locked = FALSE;
    }
```

```
}
CacheRef::CacheRef( CacheRef const & ref )
/****************************************/
{
    clone( ref );
}
CacheRef&
CacheRef::operator=(CacheRef const & ref )
/****************************************/
{
    if( this != &ref ) {
 releaseRefs();
 clone( ref );
    }
    return *this;
}
void
Cache::Add( CacheRef * ref )
/**************************/
{
    CacheChain * chain = ChainFor( ref->_name );
    ref->_chain = chain;
    chain->Latch();
    ref->_next = chain->_refs;
    chain->_refs = ref;
    chain->Unlatch();
    _assertD( ref->_next != ref );
}
void
CacheRef::force( PageInfo *info )
/*******************************/
{
    _info = info;
    _name = info->name();
    _index = 0;
    _flags = 0;
    _is.pinned_to_page = TRUE;
    _is.locked = TRUE;
    XM->Add( this );
}
void
CacheRef::force( PageInfo *info, a_page_count index )
/***************************************************/
{
    _info = info;
    _name = info->name();
    _index = index;
    _flags = 0;
    _is.locked = TRUE;
    XM->Add( this );
```

```
}
void
CacheRef::Add( a_cache_name name, a_page_count index )
/*************************************************/
{
    _name = name;
    _index = index;
    XM->Add( this );
}
CacheRef::CacheRef( a_database_number db_no, a_page_id page_no )
/*************************************************************/
{
    _info = NULL;
    _flags = 0;
    _name = NameFor( db_no, page_no );
    if( page_no == NULL_PAGE ) {
 _chain = NULL;
    } else {
 _index = 0;
 _is.pinned_to_page = TRUE;
 Add( _name );
    }
}
CacheRef::CacheRef( a_database_number db_no, a_record_id const & id )
/********************************************************************/
{
    _info = NULL;
    Set( db_no, id );
}
void
CacheRef::Set( a_database_number db_no, a_record_id const & id )
/***************************************************************/
{
    _flags = 0;
    _name = NameFor( db_no, id.page_no() );
    if( _name.as_member.page_no == NULL_PAGE ) {
 _chain = NULL;
    } else {
 Add( _name, id.index() );
    }
}
void
CacheRef::Latch( a_cache_name name, a_bool shared )
/*************************************************/
{
    _assertD( _chain == NULL );
    if( name.as_member.page_no == NULL_PAGE ) {
 releaseRefs();
    } else {
 _flags = 0;
```

```
 Add( name, 0 );
 Latch( shared );
    }
}
a_bool
CacheRef::move_to( a_page_id page_no, a_page_count index )
/*******************************************************/
// Move a reference to (page_no, index).  The reference must currently
// lock a page that contains a pointer to the new page.
{
    if( page_no == NULL_PAGE ) {
 releaseRefs();
 _index = index;
 return FALSE;
    } else {
 PageInfo *info = _info;
 _chain->Remove( this );
 Add( NameFor( info->db_no(), page_no ), index );
 _is.locked = FALSE;
 info->Unlock();
 return TRUE;
    }
}
void
CacheRef::lock_couple( a_page_id page_no, a_bool shared )
/*******************************************************/
// Set reference to (and get a read lock on) new_page_no while holding
// onto lock on existing page.
{
    PageInfo *info = _info;
    _chain->Remove( this );
    Latch( NameFor( info->db_no(), page_no ), shared );
    info->Unlock();
    mark_dirty( shared );
}
// Stubs.
void
DIO_Async_init()
/**************/
{
}
void
DIO_Async_fini()
/**************/
{
}
void
DIO_Wait_all()
/*************/
{
```

```
    // PJB FIXME:  see if we can eliminate.
    XM->WaitAll( NO_DB_ID );
}
// PJB TODO: Eliminate these routines.
void
CacheInfo::mark_dirty()
{
    ((PageInfo *) this)->mark_dirty();
}
void
CacheInfo::unlock()
{
    ((PageInfo *) this)->unlock();
}
void
CacheInfo::write_to_read_lock()
{
    ((PageInfo *) this)->ExclusiveToShared();
}
a_bool
CacheInfo::read_to_write_lock()
{
    a_bool success = ((PageInfo *) this)->TrySharedToExclusive();
    if( success ) {
 mark_dirty();
    }
    return( success );
}
void
CacheRef::unlock_page()
/*********************/
{
    Unlock();
}
////////////////////////////////////////////////////////////////////////////////
// Maintain an array of IOCBs.  Scan to find free element, if we have
// a ticket.  Claim element with atomic operation.  If we need to wait
// for an IOCB, we pick a victim and wait?  Should we keep track of
// the number of outstanding IOs?
#define IDLE_IO_DELAY 30
void
Cache::PreIO()
/************/
{
    _io.idle_timer.arm( -1 );
}
void
Cache::PostIO()
/*************/
{
```

```
    // don't need to slow this down in low power mode
    _io.idle_timer.arm( IDLE_IO_DELAY );
}
Cache::IOGlobals::IOGlobals()
/***************************/
{
    _engine_statistic_set( AVAIL_IO, MAX_IOCB );
}
a_bool
Cache::WaitAny()
/**************/
{
    for( unsigned i = 0; i < MAX_IOCB; ++i ) {
 IOCB * iocb = &_io.iocb[i];
 PageInfo * info;
 if( (info = iocb->_info) != NULL && info->TryLatch() ) {
     if( info->_pending == iocb && iocb->TryLatch() ) {
  iocb->DoFinish( info, TRUE, TRUE );
  iocb->Unlatch();
  info->Unlatch();
  return( TRUE );
     }
     info->Unlatch();
 }
    }
    return( FALSE );
}
void
Cache::WaitAll( a_database_number db_no )
/***************************************/
{
    for( unsigned i = 0; i < MAX_IOCB; ++i ) {
 IOCB * iocb = &_io.iocb[i];
 PageInfo * info;
 while( (info = iocb->_info) != NULL ) {
     if( db_no == NO_DB_ID || info->db_no() == db_no ) {
  info->Latch();
  if( db_no == NO_DB_ID || info->db_no() == db_no ) {
      if( info->_pending == iocb ) {
   iocb->Latch();
   iocb->DoFinish( info, TRUE, TRUE );
   iocb->Unlatch();
      }
  }
  info->Unlatch();
     } else {
  break;
     }
 }
    }
```

```
}
void
Cache::WaitForExtend( PageInfo * extend )
/*************************************/
// This will need to be modified if we add support for gather writes.
{
    a_database_number db_no = extend->_file->db->id();
    a_page_id page_no = extend->page_no();
   for( unsigned i = 0; i < MAX_IOCB; ++i ) {
 IOCB * iocb = &_io.iocb[i];
 PageInfo * info;
 if( (info = iocb->_info) != NULL ) {
     a_cache_name name = info->name();
     if( name.as_member.db_no == db_no && name.as_member.page_no >= page_no ) {
  if( info->Latch( this, name ) ) {
      if( info->_pending == iocb ) {
   iocb->Latch();
   iocb->DoFinish( info, TRUE, TRUE );
   iocb->Unlatch();
      }
      info->Unlatch();
  }
     }
 }
    }
}
void
Cache::WaitUntilClean( IDatabaseFile * file )
/*******************************************/
// This will need to be modified if we add support for gather writes.
{
    for( unsigned i = 0; file->dirty_pages != 0 && i < MAX_IOCB; ++i ) {
 IOCB * iocb = &_io.iocb[i];
 PageInfo * info;
 if( (info = iocb->_info) != NULL ) {
     a_cache_name name = info->name();
     if( info->file() == file ) {
  if( info->Latch( this, name ) ) {
      if( info->_pending == iocb ) {
   iocb->Latch();
   iocb->DoFinish( info, TRUE, TRUE );
   iocb->Unlatch();
      }
      info->Unlatch();
  }
     }
 }
    }
    _assertP( 101415, file->dirty_pages == 0, "File still dirty" );
}
```

```
IOCB *
Cache::GetIOCB( PageInfo * info )
/*******************************/
{
    IOCB * iocb;
    for( ;; ) {
 iocb = &_io.iocb[_io.rover++%MAX_IOCB];
 PageInfo * donor;
 if( (donor = iocb->_info) == NULL ) {
     if( iocb->TryLatch() ) {
  if( iocb->_info == NULL ) break;
  iocb->Unlatch();
     }
 } else if( donor->TryLatch() ) {
     // NB: iocb could be held while building a scatter read.
     if( donor->_pending == iocb && iocb->TryLatch() ) {
  iocb->DoFinish( donor, TRUE, TRUE );
  if( iocb->_info == NULL ) {
      donor->Unlatch();
      break;
  }
  iocb->Unlatch();
     }
     donor->Unlatch();
 }
    }
    iocb->_io_type = _taskData( io_req_type );
    iocb->_info = info;
    iocb->_mask = 1;
    _engine_statistic_add( AVAIL_IO, -1 );
    // PJB TODO:
    return( iocb );
}
a_bool
IOCB::Finish( PageInfo * info, a_bool wait, a_bool respect_evict )
/*****************************************************************/
{
    a_bool success = DoFinish( info, wait, respect_evict );
    Unlatch();
    return( success );
}
a_bool
IOCB::Finish( a_bool wait, a_bool respect_evict )
/***********************************************/
{
    // PJB FIXME: what about latching _info?
    return( Finish( _info, wait, respect_evict ) );
}
a_bool
IOCB::DoFinish( PageInfo * info, a_bool wait, a_bool respect_evict )
```

```
/*****************************************************************/
// Might have "shared" access to page if waiting for shared access to
// page being written.
{
   _assertD( info->_pending == this );
   if( !wait_for_io( Task::me(), wait ) ) {
 _assertD( !wait );
 return( FALSE );
   }
   if( IsWrite() ) {
 _assertD( info == _info );
 if( Waited() ) {
    _db_statistic_incr( db, NULL, DISK_WAITWRITE );
 }
 info->w_complete();
 if( _is.to_be_evicted && respect_evict ) {
    // must have exclusive access if we are to evict
    _assertD( info->HaveExclusively() );
    XM->RemoveFromHash( info );
    _is.to_be_evicted = FALSE;
 }
 _info = NULL; // Gather writes not supported.
   } else {
 if( Waited() ) {
    _db_statistic_incr( db, NULL, DISK_WAITREAD );
    switch( _io_type ) {
    case IOREQ_OPTIM :
 _db_statistic_incr( db,NULL,WAITREAD_OPTIM );    break;
    case IOREQ_TEMPTAB :
 _db_statistic_incr( db,NULL,WAITREAD_TEMPTAB );  break;
    case IOREQ_FULLCOMP :
 _db_statistic_incr( db,NULL,WAITREAD_FULLCOMP ); break;
    default  :
 _db_statistic_incr( db,NULL,WAITREAD_UNKNOWN );
    }
 }
 info->r_complete();
 if(  _mask == 1 ) {
    _info = NULL;
 } else if( info == _info ) {
    a_cache_name name = info->name();
    do {
 name.as_member.page_no++;
 _mask >>= 1;
   } while( (_mask&1) == 0 );
    _info = XM->ChainFor( name )->AssuredFind( name );
 } else {
    _mask &= ~(1 << (info->page_no() - _info->page_no()));
 }
   }
```

```
    if( _info == NULL ) {
 _engine_statistic( AVAIL_IO );
    }
    // A failed IO may have completed on another thread
    SuicideOnFatalError();
    // PJB FIXME:   should we pass pointer to wait
    // counter?  Guaranteed that only one waiter from
    // upper levels (owner of IOCB). Should do wait_for_io( me ).
    return( TRUE );
}
a_bool
IOCB::IsActuallyPending( void )
/*****************************/
{
#if defined( UNDER_CE )
    return FALSE;
#elif defined( WINNT )
    return _state.pending_io;
#else
    return _state.as_member.pending_io;
#endif
}
void
Cache::AddToReusable( PageInfo * info )
/*************************************/
{
    if( !info->IsMissingImage() ) {
 _assertD( info->HaveExclusively() && !info->IsInHash() );
 info->_name.as_uint = BOGUS_NAME;
 if( info->_pending != (IOCB *) PageInfo::REUSABLE ) {
     _assertD( info->_pending == NULL );
     info->_pending = (IOCB *) PageInfo::REUSABLE;
 }
#if SUPPORT_CACHE_COLDSTART
 info->SetColdstartRecorded( FALSE );
#endif
 _reusable.Enqueue( info );
    }
}
PageInfo *
Cache::AllocReusable()
/*********************/
{
    PageInfo * info = _reusable.Dequeue();
    if( info && info->TryLatch() ) {
 if( info->IsReusable() ) {
     info->_pending = NULL;
     return info;
 }
 info->Unlatch();
```

```
    }
    return( NULL );
}
// PageInfo methods.
PageInfo::~PageInfo()
/*******************/
{
}
PageInfo::PageInfo( void * image )
/*********************************/
    : _next( NULL )
    , _flags( 0 )
#if !PRODUCTION
    , first_lock( 0 )
    , first_write( 0 )
#endif
    , _real_page_no( 0 )
#if defined( CHECKSUM )
    , checksum( 0 )
#endif
    , _pending( (IOCB *) IMAGE_MISSING )
{
    _name.as_uint = BOGUS_NAME;
    _image = image;
    _is.preimage_may_not_be_committed = FALSE;
#ifdef SUPPORT_CACHE_COLDSTART
    _is.coldstart_recorded = FALSE;
#endif
}
a_bool
PageInfo::CanPreventReuse()
/*************************/
{
    a_name_reuse_state state = _state;
    a_name_reuse_state new_state;
    while( state.bits.is_in_hash ) {
 new_state = state;
 new_state.bits.ref_count++;
 if( CSwap( &_state.as_cell, &state.as_cell, new_state.as_cell ) ) return TRUE;
    }
    return FALSE;
}
a_bool
PageInfo::AllowReuse()
/********************/
{
    a_name_reuse_state state = _state;
    a_name_reuse_state new_state;
    do {
 new_state = state;
```

```
 new_state.bits.ref_count--;
    } while( !CSwap( &_state.as_cell, &state.as_cell, new_state.as_cell ) );
    return !state.bits.is_in_hash;
}
a_bool
PageInfo::SetInHash( a_bool in_hash )
/***********************************/
{
    a_name_reuse_state state = _state;
    a_name_reuse_state new_state;
    do {
 new_state = state;
 new_state.bits.is_in_hash = in_hash;
    } while( !CSwap( &_state.as_cell, &state.as_cell, new_state.as_cell ) );
    return new_state.bits.ref_count == 0;
}
void
PageInfo::RecordHit( Cache * cm )
/*******************************/
{
    unsigned now = cm->_now;
    if( (int) (now -_visited_at) > cm->_delta ) {
 _visited_at = now;
 a_prs_slot score = _score;
 _score = score >= MAX_SCORE? MAX_SCORE : score + 1;
    }
}
a_bool
PageInfo::Latch( Cache * cm, a_cache_name name, a_bool shared )
/*************************************************************/
// Block only for pages named name.  Returns TRUE if latch has been
// obtained (FALSE => the name of the info might have changed).
{
    if( TryLatch( shared ) ) {
 if( HasName( name ) ) {
     RecordHit( cm );
     return( TRUE );
 }
 Unlatch();
    } else if( HasName( name ) ) {
 if( CanPreventReuse() ) {
     a_bool recycle;
     if( HasName( name ) ) {
  Latch( shared );
  recycle = AllowReuse();
  if( HasName( name ) ) {
      _assertD( !recycle );
      RecordHit( cm );
      return( TRUE );
  }
```

```
  if( recycle && !IsPinned() ) {
      if( !shared || TrySharedToExclusive() ) {
   cm->AddToReusable( this );
      }
  }
  Unlatch();
     } else {
  // We just lost a race trying to latch page
  // so let the cleaner pick up the page.
  AllowReuse();
     }
 }
    }
    return( FALSE );
}
a_bool
PageInfo::Scrub()
/***************/
{
#if !PRODUCTION
    class IOCB * pending = _pending;
    _assertD( pending == NULL && HaveExclusively() );
#endif
    if( _is.tracked_table_page ) {
 _file->db->_pagectr->RemoveCachedPage( _tid, page_no(), !_is.blob_page );
 _is.tracked_table_page = FALSE;
    }
    if( _is.dirty ) {
 _is.dirty = FALSE;
 _is.preimage_may_not_be_committed = FALSE;
 if( !_is_temporary( _file->origin ) ) {
     _file->dec_dirty_pages();
 }
 return( TRUE );
    }
    return( FALSE );
}
a_bool
Cache::RemoveFromHash( PageInfo * info )
/**************************************/
{
    CacheChain * chain = ChainFor( info->name() );
    a_bool was_dirty = info->Scrub();
    chain->Remove( info );
    info->_name.as_uint = BOGUS_NAME;
    a_bool reusable = info->SetInHash( FALSE );
    if( reusable ) {
 AddToReusable( info );
    }
    return( was_dirty );
```

```
}
void
PageInfo::PrepareForAdd( a_cache_name name, unsigned now )
/*******************************************************/
// Prepare page for (possibly) going into cache.
{
    _visited_at = now;
    _score = 0;
    _name = name;
    _file = NULL;
    _assertD( !_is.tracked_table_page );
    _real_page_no = name.as_member.page_no;
}
PageInfo *
Cache::AddToHash( CacheChain * chain, a_cache_name name, PageInfo * alloc )
/*************************************************************************/
{
    alloc->PrepareForAdd( name, _now );
    PageInfo * info = chain->FindOrInsert( name, alloc );
    if( info == alloc ) {
 info->SetInHash( TRUE );
 info->_pending = (IOCB *) PageInfo::JUST_INSTALLED;
 ++_now;
    }
    return( info );
}
inline a_bool
PageInfo::IsReapable() const
/*******************/
{
    return !_pending;
}
void
PageInfo::DoWaitForPending()
/**************************/
{
    _assertD( !_is.preimage_may_not_be_committed );
    TRACE( MultiSpindleWaitPending( 0 ) );
    IOCB * iocb = _pending;
    if( iocb != NULL ) {
 iocb->Latch();
 if( _pending == iocb ) {
     iocb->DoFinish( this, TRUE, FALSE );
 }
 iocb->Unlatch();
    }
}
void
PageInfo::PrepareForRead( IOCB * iocb )
/*************************************/
```

```
{
    Database * db = _CurrentDB;
    // need to prepare for reading if shared
    _file = file_for_page( db, page_no() );
    a_physical_page location = page_no() - _file->origin;
    if( location > db->DB_Max_physical_page_number ) {
 DB_Fatal( SQLSTATE_ACCESS_BEYOND_END_OF_MAX_DBSPACE,
   IDS_ENG_FMSG_ACCESS_BEYOND_END_OF_MAX_DBSPACE );
    }
    _real_page_no = page_no();
    _assertD( !_is.preimage_may_not_be_committed );
    _pending = iocb;
    _grant_image_access();
#if !PRODUCTION
    if( !db->is_magic_db() ) {
 db->Store->ClearPageNo( _image );
    }
#endif
}
IOCB *
PageInfo::StartRead()
/*******************/
{
    PrepareForRead( XM->GetIOCB( this ) );
    _file->start_read( this );
    return( _pending );
}
void
PageInfo::DoWaitForPreimage()
/***************************/
{
    Database * db = _file->db;
    if( db->has_separate_checkpoint_log() ) {
 // A preimage has been added to the (separate) checkpoint log
 // and may or may not have been committed yet.
 _assertD( _checkpoint_log_pos != 0 );
 if( _checkpoint_log_pos >
     _file->db->_checkpoint_log->last_committed_pos() ) {
     // Force the preimage to be committed
     _file->db->_checkpoint_log->flush( _checkpoint_log_pos );
 }
    } else {
 // Preimage is in database file:  if it hasn't been committed
 // it will be in cache.
 PageInfo * info = XM->IsInCache( NameFor( db_no(), _rollback_page_no ) );
 if( info != NULL ) {
     info->WaitForPending();
     info->Unlatch();
 }
 // Ensure that the preimage has been committed by OS.
```

```
 _file->flush_fsys_cache();
    }
    _is.preimage_may_not_be_committed = FALSE;
}
IOCB *
PageInfo::StartWrite( a_bool evict )
/*********************************/
// Must have exclusive access going in.
{
    SuicideOnFatalError();
    _CurrentConnection->AddIOCount();
    WaitForPreimage();
    _pending = XM->GetIOCB( this );
    _pending->_is.to_be_evicted = evict;
    _grant_image_access();
    _file->start_write( this );
    _assertD( !_is.preimage_may_not_be_committed );
    return( _pending );
}
void
PageInfo::Write( a_bool wait )
/****************************/
{
    StartWrite()->Finish( wait, FALSE );
}
void
PageInfo::WriteAndEvict()
/***********************/
{
    StartWrite( TRUE )->Finish( FALSE, TRUE );
    Unlatch();
}
void
PageInfo::Evict()
/***************/
{
    WaitForPending();
    WaitForPreimage();
    XM->RemoveFromHash( this );
    Unlatch();
}
void
PageInfo::write_and_flush()
/*************************/
{
    Write( TRUE );
    _file->flush_fsys_cache();
}
a_bool
PageInfo::is_pending()
```

```
/********************/
{
    if( _pending == (IOCB *) REUSABLE
 || _pending == (IOCB *) IMAGE_MISSING
 || _pending == (IOCB *) PINNED ) {
 //this should never happen to pages we are interested in
 _assertD( FALSE );
 return TRUE;
    } else {
 return _pending != NULL && _pending->IsActuallyPending();
    }
}
void
PageInfo::FinishLock( a_bool shared )
/***********************************/
{
    Worker * me = _CurrentWorker;
    Database   *db = me->active_db();
    Connection *c = me->active_con();
    _db_statistic_incr( db, c, CACHE_READ );
    Eng->cache_stats->_lookups++;
    _engine_statistic( CACHE_READ_ENG );
#ifdef SUPPORT_CACHE_COLDSTART
    if( !IsColdstartRecorded() && (_real_page_no == page_no() ) ) {
        SetColdstartRecorded( TRUE );
        db->_coldstart_manager.record_page( _real_page_no );
    }
#endif
    if( JustInstalled() ) {
 _assertD( HaveExclusively() && IsAddressable() );
 StartRead()->Finish( TRUE, FALSE );
 if( shared ) {
     ExclusiveToShared();
 }
 _assertD( _pending == NULL );
    } else {
 WaitForPending();
 _db_statistic_incr( db, c, CACHE_HITS );
 Eng->cache_stats->_cache_hits++;
 _engine_statistic( CACHE_HITS_ENG );
 _assertD( _pending == NULL );
 XM->MakeAddressable( this );
    }
#if !PRODUCTION
    if( !DisableAssertNotFree ) {
 AssertNotFree( db );
    }
#endif
    // Since async IOs can complete on another task,
    // a failed IO may have caused the assertion on that
```

```
    // other task. We cannot allow the caller to continue
    // because it may assume that all IOs have completed
    // successfully.
    SuicideOnFatalError();
    _grant_image_access();
    TRACE( CacheLock( _name.as_uint ) );
    switch( _file->get_page_usage( _image ) ) {
    case TABLE_PAGE:
 _db_statistic_incr( db, c, CACHE_TABLE_READ );
 break;
    case INDEX_PAGE:
 _db_statistic_incr( db, c, CACHE_INDEXLEAF_READ );
 break;
    }
}
void
PageInfo::unlock()
/****************/
{
    TRACE( CacheUnlock( _name.as_uint ) );
    _revoke_image_access();
    Unlatch();
}
void
PageInfo::AssertNotFree( Database * db )
/*************************************/
{
    if( db->has_free_page_bit_maps() ) {
 _assertD( _file->chkpt_free_map == NULL
  || !_file->chkpt_free_map->IsMember( page_no() ) );
 _assertD( _file->current_free_map == NULL
  || !_file->current_free_map->IsMember( page_no() ) );
 _assertD( _file->reusable_free_map == NULL
  || !_file->reusable_free_map->IsMember( page_no() ) );
    }
}
void
PageInfo::r_complete()
/********************/
{
    _file->finish_read( this );
    XM->assert_page_no( _file->db, page_no(), _image );
    _pending = NULL;
    _revoke_image_access();
}
void
PageInfo::w_complete()
/********************/
{
    _assertD( !_is.preimage_may_not_be_committed );
```

```
    _file->finish_write( this );
    if( _is.dirty ) {
 // Might not be dirty during cache flushes, etc.
 _is.dirty = FALSE;
 if( !_is_temporary( _file->origin ) ) {
     _file->dec_dirty_pages();
     _file->db->_auto_ckpt->inc_recovery_io();
 }
    }
    _pending = NULL;
    _revoke_image_access();
}
// PJB FIXME.
void
CacheInfo::SetTableInfo( a_table_id tid, a_bool is_blob_page )
/*************************************************************/
{
    PageInfo * info = (PageInfo *) this;
    if( tid != NO_TABLE_ID ) {
 info->_tid = tid;
 info->_is.blob_page = is_blob_page;
 info->_is.tracked_table_page = TRUE;
 AddCachedPage( db(), tid, page_no(), !is_blob_page );
    }
}
void
CacheInfo::Pin()
/**************/
{
    ((PageInfo *) this)->Pin();
}
void
CacheInfo::Unpin()
/****************/
{
    ((PageInfo *) this)->Unpin();
}
void
PageInfo::SetPreImage( a_page_id page_no )
/****************************************/
{
    _is.preimage_may_not_be_committed = TRUE;
    _checkpoint_log_pos = page_no;
}
// CacheChain methods.
void
CacheChain::Add( CacheRef * ref )
/*******************************/
{
    Latch();
```

```
    ref->_next = _refs;
    _refs = ref;
    ref->_chain = this;
    Unlatch();
}
a_bool
CacheChain::Remove( CacheRef * remove )
/*************************************/
{
    a_bool right_chain;
    Latch();
    if( remove->_chain == this ) {
 CacheRef * ref;
 CacheRef ** pref;
 for( pref = &_refs; (ref = *pref) != NULL; pref = &ref->_next ) {
     if( ref == remove ) {
  *pref = ref->_next;
  ref->_chain = NULL;
  Unlatch();
  return TRUE;
     }
 }
 right_chain = TRUE;
    } else {
 right_chain = FALSE;
    }
    Unlatch();
    return right_chain;
}
// Hash lookup table manipulation
PageInfo *
CacheChain::Find( a_cache_name name )
/***********************************/
{
    a_cache_name last;
    last.as_uint = 0;
    for( PageInfo * info = _head; info != NULL; info = info->_next ) {
 a_cache_name current_name = info->_name;
 if( current_name.as_uint == name.as_uint ) return( info );
 if( current_name.as_uint >  name.as_uint ) return( NULL );
 if( current_name.as_uint <  last.as_uint ) return( NULL );
 // PJB FIXME:  Is this safe?
 last.as_uint = current_name.as_uint + 1;
    }
    return( NULL );
}
PageInfo *
CacheChain::AssuredFind( a_cache_name name )
/******************************************/
{
```

```
    PageInfo * info = Find( name );
    if( info == NULL ) {
 LatchInfos();
 info = Find( name );
 UnlatchInfos();
    }
    return( info );
}
PageInfo *
CacheChain::FindOrInsert( a_cache_name name, PageInfo * insert )
/*****************************************************************/
{
    LatchInfos();
    PageInfo * info;
    PageInfo ** pinfo;
    for( pinfo = &_head; (info = *pinfo) != NULL; pinfo = &info->_next ) {
 if( info->_name.as_uint >= name.as_uint ) {
     if( info->_name.as_uint == name.as_uint ) goto done;
     break;
 }
    }
    insert->_next = info;
    *pinfo = info = insert;
done:
    UnlatchInfos();
    return info;
}
void
CacheChain::Remove( PageInfo * remove )
/*************************************/
{
    LatchInfos();
    PageInfo * info;
    PageInfo ** pinfo;
    for( pinfo = &_head; (info = *pinfo) != NULL; pinfo = &info->_next ) {
 if( info == remove ) {
     *pinfo = info->_next;
     break;
 }
    }
    UnlatchInfos();
}
// Basic cache operations.
PageInfo *
Cache::Install( a_cache_name name, a_bool shared )
/************************************************/
{
    CacheChain * chain = ChainFor( name );
    PageInfo   * info;
    for( info = chain->Find( name );; ) {
```

```
 if( info && info->Latch( this, name, shared ) ) break;
 PageInfo * alloc = Alloc();
 info = AddToHash( chain, name, alloc );
 if( info == alloc ) break;
 AddToReusable( alloc );
 alloc->Unlatch();
    }
    _assertD( chain->AssuredFind( name ) == info );
    return info;
}
CacheInfo *
Cache::Install( a_database_file * f, a_page_id page_no, a_page_id real_page_no )
/*****************************************************************************/
{
    a_cache_name name = NameFor( f->db->id(), page_no );
    PageInfo * info = Install( name );
    if( real_page_no != page_no ) {
 f = file_for_page( f->db, real_page_no );
    }
    _assertD( info->_name.as_uint == name.as_uint );
    if( info->JustInstalled() ) {
 info->_pending = NULL;
 info->_file = f;
 info->_real_page_no = real_page_no;
    } else {
 // PJB FIXME: should be assertPs
 _assertD( info->_file == f );
 _assertD( info->_real_page_no == real_page_no );
 // PJB FIXME:  How can we get here?
 // _assertD( !info->is_dirty() );
 info->WaitForPending();
 a_bool was_dirty = info->Scrub();
 _unused( was_dirty );
 MakeAddressable( info );
    }
    // PJB FIXME: cf force_hash
    _grant_image_access();
    return( info );
}
PageInfo *
Cache::IsNotInCache( a_cache_name name )
/**************************************/
{
    CacheChain * chain = ChainFor( name );
    PageInfo * info = chain->Find( name );
    if( info == NULL ) {
 PageInfo * alloc = Alloc();
 info = AddToHash( chain, name, alloc );
 if( info == alloc ) return info;
 AddToReusable( alloc );
```

```
 alloc->Unlatch();
    }
    return NULL;
}
PageInfo *
Cache::IsInCache( a_cache_name name )
/***********************************/
{
    PageInfo * info = ChainFor( name )->AssuredFind( name );
    if( info && info->TryLatch() ) {
 if( info->HasName( name ) ) return info;
 info->Unlatch();
    }
    return NULL;
}
PageInfo *
Cache::LatchIfInCache( a_cache_name name )
/****************************************/
{
    PageInfo * info = IsInCache( name );
    if( info ) {
 info->WaitForPending();
 MakeAddressable( info );
    }
    return info;
}
a_bool
Cache::IsImmediatelyLatchable( a_cache_name name )
/************************************************/
{
    PageInfo * info = ChainFor( name )->AssuredFind( name );
    return info != NULL && !info->is_pending();
}
a_bool
Cache::IsIOPending( a_cache_name name )
/*************************************/
{
    PageInfo * info = ChainFor( name )->AssuredFind( name );
    return info != NULL && info->is_pending();
}
// Page replacement routines.
PageInfo *
Cache::Alloc()
/************/
{
    PageInfo * info;
    unsigned fence = _rover + avail_infos();
    while( (info = AllocReusable()) == NULL ) {
 if( (int) (fence - _rover) < 0 || (info = Scavenge()) == NULL ) {
     while( info == NULL ) {
```

```
 info = WaitAny()? Scavenge() : Panic();
    }
    fence = _rover + avail_infos();
 }
 _assertD( info->HaveExclusively() );
 if( info->IsDirty() ) {
    info->StartWrite( TRUE )->Finish( FALSE, TRUE );
    // PJB TODO: Should we wait if too many pending IOs?
 } else {
    if( info->IsInHash() ) {
  Eng->cache_stats->_cache_replacements++;
  _engine_statistic( CACHE_REPLACEMENTS );
  info->WaitForPending();
  RemoveFromHash( info );
    } else {
  AddToReusable( info );
    }
 }
 info->Unlatch();
   }
   MakeAddressable( info );
   _assertD( !info->_is.dirty );
#if defined( LINUX )
   // Linux has VM manager bug in kernels before 2.2.17.
   // An attempt to write to an OS page that is not in memory
   // corrupts first 16 bytes of that page. Work around was to
   // touch first byte of the OS page so that it gets allocated into
   // the process memory map.
   if( _touch_page_before_use ) {
  _grant_image_access();
      char *ptr = (char *) info->_image;
      char *end_of_page = ptr + _db_page_size_max;
      while( ptr < end_of_page ) {
         ((int*)ptr)[0] = 0;
         ptr += _os_page_size;
      }
  _revoke_image_access();
   }
#endif
   return( info );
}
void
PageInfo::Pin()
/*************/
{
   _assertD( HaveExclusively() );
   _pending = (IOCB *) PINNED;
   ++XM->_pinned_images;
   _engine_statistic_set( CACHE_PINNED, XM->_pinned_images );
   TRACE( CachePin( _name.as_uint ) );
```

```
    Unlatch();
}
void
PageInfo::Unpin()
/***************/
{
    Latch();
    _assertD( _pending == (IOCB *) PINNED );
    TRACE( CacheUnpin( _name.as_uint ) );
    _pending = NULL;
    --XM->_pinned_images;
    _engine_statistic_set( CACHE_PINNED, XM->_pinned_images );
}
void *
Cache::AllocImage( CacheInfo ** pinfo )
/************************************/
// Exclusive access, but might be freed by another task.
{
    PageInfo *info = Alloc();
    *pinfo = info;
    info->Pin();
    _grant_image_access();
    return( info->_image );
}
void
Cache::FreeImage( CacheInfo * cinfo )
/***********************************/
{
    PageInfo * info = (PageInfo *) cinfo;
    _revoke_image_access();
    info->Unpin();
    AddToReusable( info );
    info->Unlatch();
}
void *
Cache::multi_page_alloc( uint32  num_pages,
    uint32  page_size,
    MultiPageAlloc *handle )
/*********************************************/
// Alloc a group of contiguous pages, aligned on a page boundary
// (or on a sector boundary on NT).
{
    // First try to allocate from the cache
    // if( successful ) {
    //  handle->_aligned = (starting page);
    // handle->_malloced = FALSE;
    // } else {
 #if defined( WINNT )
     handle->_unaligned_mem = NULL;
     handle->_aligned_mem = VirtualAlloc( NULL, page_size * num_pages,
```

```
      MEM_COMMIT, PAGE_READWRITE );
 #else
    handle->_unaligned_mem = ut_alloc( (num_pages + 1) * page_size );
    handle->_aligned_mem = align( handle->_unaligned_mem, page_size );
 #endif
 _assertP( 101413, handle->_aligned_mem != NULL,
   "Unable to allocate a multi-page block of memory" );
 handle->_malloced = TRUE;
 #if !PRODUCTION
 {
    mp_alloc  *mpalloc;
    mpalloc = (mp_alloc *)ut_alloc( sizeof( mp_alloc ) );
    mpalloc->contents = handle->_aligned_mem;
    mpalloc->next = MultiPageAllocs;
    ++MultiPageCount;
    mpalloc->count = MultiPageCount;
    MultiPageAllocs = mpalloc;
    if( MultiPageCount == WMPmem ) {
  BMPmem( WMPmem );
    }
 }
 #endif
 return( handle->_aligned_mem );
   // }
}
void
Cache::multi_page_free( MultiPageAlloc *handle )
/*********************************************/
{
   #if !PRODUCTION
 {
    mp_alloc  **head;
    mp_alloc  *mpalloc;
    if( handle->_aligned_mem != NULL ) {
  for( head = (mp_alloc **)&MultiPageAllocs;; ) {
     mpalloc = *head;
     _assertD( mpalloc != NULL );
     if( mpalloc->contents == handle->_aligned_mem ) {
   if( mpalloc->count == WMPmem ) {
      BMPmem( WMPmem );
   }
   *head = mpalloc->next;
   ut_free( mpalloc );
   break;
     }
     head = &mpalloc->next;
  }
    }
 }
   #endif
```

```
    if( handle->_malloced ) {
 #if defined( WINNT )
     VirtualFree( handle->_aligned_mem, 0, MEM_RELEASE );
 #else
     ut_free( handle->_unaligned_mem );
 #endif
 handle->_malloced = FALSE;
 handle->_unaligned_mem = NULL;
 handle->_aligned_mem = NULL;
    } else {
 // Release pages allocated from the cache.
    }
}
class ScavengeState {
public:
    ScavengeState()
 : threshold( 0 )
 , counter( 0 )
    {
    }
public:
    unsigned threshold;
    unsigned counter;
};
#if defined( AWE_CACHE )
void
Cache::DoMakeAddressable( PageInfo * info )
/*****************************************/
// NOTE: We never steal the address space of a dirty page
// so that don't need to go looking for address space at checkpoint
// time
{
    PageInfo ** frame;
    PageInfo * owner;
    a_byte * image;
    ScavengeState state;
    for( a_ptrint fence = _frame_rover + _awe_cache._addr_space;; ) {
 if( info->_image != NULL ) {
     frame = NULL;
     break;
 }
 a_ptrint rover = _frame_rover++;
 frame = &_frame_owner[rover%_awe_cache._addr_space];
 owner = *frame;
 if( owner == NULL ) {
     if( CSwapPtr( frame, &owner, info ) ) {
  image = ordinal_to_image( frame - _frame_owner );
  break;
     }
 } else if( CanScavenge( owner, &state ) ) {
```

```
    if( owner->IsDirty() ) {
 owner->StartWrite( TRUE )->Finish( FALSE, TRUE );
 // PJB TODO: Should we wait if too many pending IOs?
    } else if( owner->_image != NULL ) {
 image = (a_byte *) owner->_image;
 frame = &_frame_owner[_images.image_to_ordinal( image )];
 break;
    }
    owner->Unlatch();
}
if( (int) (fence - rover) < 0 ) {
    if( !WaitAny() ) {
 DB_Fatal( SQLSTATE_DYNAMIC_MEMORY_EXHAUSTED, NULL );
    }
    fence += _awe_cache._addr_space;
}
  }
  void * old = info->_image;
  if( frame != NULL ) {
for( ;; ) {
    if( old != NULL ) {
 if( owner != NULL ) {
     owner->Unlatch();
 } else {
     *frame = NULL;
 }
 break;
    }
    if( CSwapPtr( &info->_image, &old, TAG_UNMAPPED( image ) ) ) {
 if( owner != NULL ) {
     *frame = info;
     owner->_image = NULL;
     owner->Unlatch();
 }
 break;
    }
}
  }
  image = STRIP_TAG( info->_image );
  while( info->_image != image ) {
if( _awe_cache.map_image( image, ordinal( info ) ) ) {
    while( !CSwapPtr( &info->_image, &old, image ) );
    break;
}
  }
}
#endif
a_bool
Cache::CanScavenge( PageInfo * info, ScavengeState * state )
/*************************************************************/
```

```
{
    if( info->IsReusable() && !info->IsLatched() ) {
 if( info->TryLatch() ) {
     if( info->IsReusable() ) {
  return TRUE;
     }
     info->Unlatch();
 }
    } else if( info->IsReapable() && !info->IsInUse() ) {
 unsigned score = (14*info->_score)/16;
 if( score > state->threshold ) {
     info->_score = score;
     if( state->counter == 0 ) {
  if( state->threshold < MAX_THRESHOLD ) {
      ++state->threshold;
  }
  state->counter = state->threshold*_incr;
     } else {
  --state->counter;
     }
 } else if( info->TryLatch() ) {
     if( info->IsReapable() && !info->IsPinned() ) {
  return TRUE;
     }
     info->Unlatch();
 }
    }
    return FALSE;
}
PageInfo *
Cache::Scavenge()
/***************/
// Attempt to find a page (not necessarily with address space) suitable
// for reuse.
{
    ScavengeState state;
    unsigned  i = _rover;
    unsigned  j;
    while( (unsigned) (j = _rover++ - i) < avail_infos() ) {
 // PJB FIXME: get rid of mod
 PageInfo * info = &_info[(i+j)%avail_infos()];
 if( CanScavenge( info, &state ) ) {
     return info;
 }
    }
    return NULL;
}
PageInfo *
Cache::Panic()
/************/
```

```
{
#if defined( DYNAMIC_CACHE_SIZE )
    // We must be absolutely certain that it is impossible to grow
    // the cache and that we didn't fail to scavenge something just
    // because someone else shrank the cache or grew it to its max
    // after we did our first scavenging.
    //
    // However, it is possible to enter the panic code while growing
    // the cache so avoid recursion if we are growing the cache.
    _cache_growth_mutex.get();
#endif
#if 0
    // DT_Find_memory is currently stubbed out.  We can safely returned if
    // we have returned pages.  Calling Scavenge will not tell us that.
    DT_Find_memory();
    PageInfo * info = Scavenge();
#endif
    PageInfo * info = NULL;
#if defined( DYNAMIC_CACHE_SIZE )
    #define PANIC_GROWTH_AMOUNT  64
    if( !info && grow_cache_by_images( PANIC_GROWTH_AMOUNT ) ) {
 info = Scavenge();
    }
    _cache_growth_mutex.give();
#endif
    if( !info ) {
 DB_Fatal( SQLSTATE_DYNAMIC_MEMORY_EXHAUSTED,
    IDS_ENG_FMSG_DYNAMIC_MEMORY_EXHAUSTED );
    }
    return( info );
}
// User routines.
PageInfo *
Cache::Lock( a_cache_name name, a_bool shared )
/*********************************************/
{
    PageInfo * info = LockRaw( name, shared );
    assert_page_no( info->_file->db, info->page_no(), info->_image );
    // FIXME: JCS
    // info->_file->sanity_check( page_no, info->_image, TRUE );
    return( info );
}
// PJB FIXME: other lock functions that should be implemented as wrappers that
// call lock and markdirty as needed.
// Note that we cannot convert read to write latch without the possibility
// of giving up info.
/* PJB FIXME:  Implement
void
PageInfo::Unlock()
{
```

```
//    _debug( DbgPageUnlock( this ) );
    Unlatch();
    if( !latched ) {
 _vm_protect( this );
    }
}
*/
unsigned
Cache::Hint( a_cache_name name )
/******************************/
{
    PageInfo * info = IsNotInCache( name );
    if( info == NULL ) {
 return( 0 );
    }
    info->StartRead()->Finish( FALSE, FALSE );
    info->Unlock();
    _db_statistic_incr( DBFromID( name.db_no ), NULL, READ_HINTS );
    return( 1 );
}
void
Cache::Evict( a_cache_name name )
/******************************/
// For FreePageById
{
    PageInfo * info = IsInCache( name );
    if( info != NULL ) {
 info->Evict();
    }
}
void
Cache::Evict( a_database_number db_no, unsigned file_no, a_bool is_scrammed )
/*****************************************************************************/
// Checkpoint must be done beforehand.
{
#if PRODUCTION
    _unused( is_scrammed );
#endif
    for( unsigned i = 0; i < avail_infos(); ++i ) {
 PageInfo * info = &_info[i];
 a_cache_name name = info->name();
 if( name.as_member.db_no == db_no &&
     ( file_no == ALL_FILES
      || file_no == _file_num( name.as_member.page_no )
      || (file_no == MAPPED_PAGES
  && info->page_no() != info->_real_page_no) ) ) {
     if( info->Latch( this, name ) ) {
  if( info->IsPinned() ) {
      // Heap was left locked:  track it down.
      _assertPNR( 101416, FALSE, "Heap left locked at database close." );
```

```
      info->_pending = NULL;
    } else {
      info->WaitForPending();
    }
    a_bool was_dirty = RemoveFromHash( info );
    _unused( was_dirty ); // for PRODUCTION builds
    _assertD( !was_dirty || is_scrammed || _is_temporary( name.as_member.page_no ) );
    info->Unlatch();
      }
   }
     }
}
struct a_flushed {
    PageInfo *  info;
    a_page_id  page_no;
};
static int
compare_page_ids( const void *elem1, const void *elem2 )
/*******************************************************/
{
    // Cannot just return the difference between the two page numbers
    // because the page numbers are unsigned and can have the high bit
    // set if dbspace 8 or higher exists. That screws up the ordering
    // and the external checkpoint log relies on it.  See flush_pages.
    a_page_id  pg1 = ((a_flushed *)elem1)->page_no;
    a_page_id  pg2 = ((a_flushed *)elem2)->page_no;
    return( (pg1 == pg2) ? 0 : ((pg1 < pg2) ? -1 : 1) );
}
void
Cache::FlushPages( Database *db, a_flushed *flushed, unsigned n )
/*****************************************************************/
// Flush all marked pages for specified database.
{
    // NOTE: When using an external checkpoint log, there is an
    // implicit assumption that the following qsort() is done. If the
    // definition page for the main dbspace is an element of the
    // "infos" set, we must start a write on that definition page
    // before starting an IO on any other page in the set because
    // starting a write on any other page may cause the separate
    // checkpoint log code to attempt to write-lock the definition
    // page when flushing a preimage.  Since the "writing" bit is
    // already turned by Cache::flush for all of the pages we are
    // about to write, write-locking the definition page will cause
    // the write-lock by the separate checkpoint log code to hang
    // waiting for the write of the definition page to complete (an IO
    // which has not actually been issued yet).
    qsort( flushed, n, sizeof(flushed[0]), compare_page_ids );
    _taskData( io_req_type ) = IOREQ_CHKPTWRT;
    for( unsigned i = 0; i < n; ++i ) {
  a_cache_name name = NameFor( db->id(), flushed[i].page_no );
```

```
 PageInfo * info = flushed[i].info;
 if( info->Latch( this, name ) ) {
    if( info->_is.dirty ) {
  info->WaitForPending();
  if( info->_is.dirty ) {
      _checksum( info->_image );    // blank padding
      info->StartWrite()->Finish( FALSE, TRUE );
  }
    }
    info->Unlatch();
 }
   }
   _taskData( io_req_type ) = IOREQ_UNKNOWN;
   _db_only_statistic_add( db, CHECKPOINT_FLUSH, n );
}
#define FLUSH_MAX 256
#if !defined(ON_BUILD_MACHINE) && defined( WINNT ) && !defined( UNDER_CE )
#define MEASURE_WRITE_TIMES
#endif
void
Cache::DoFlush( p_database db, a_bool flush_temp )
/***********************************************/
{
   a_flushed flushing[FLUSH_MAX];
   unsigned flush_count = 0;
#if defined(MEASURE_WRITE_TIMES)
   unsigned flush_total = 0;
   LARGE_INTEGER start_time;
   LARGE_INTEGER stop_time;
   LARGE_INTEGER frequency;
   QueryPerformanceCounter( &start_time );
#endif
   a_database_number db_no = db->id();
   _assertP( 101403, _CurrentWorker->is_forbidding(),
      "FlushCache: worker is not forbidding" );
   for( a_ptrint i = 0; i < avail_infos(); i++ ) {
 PageInfo * info = &_info[i];
 a_cache_name name = info->name();
 if( name.as_member.db_no == db_no && info->_is.dirty &&
    (flush_temp || !_is_temporary( name.as_member.page_no )) &&
    !info->IsPinned() ) {
    flushing[flush_count].info = info;
    flushing[flush_count].page_no = name.as_member.page_no;
    ++flush_count;
 }
 if( flush_count == FLUSH_MAX ) {
    FlushPages( db, flushing, flush_count );
#if defined(MEASURE_WRITE_TIMES)
    flush_total += flush_count;
#endif
```

```
     flush_count = 0;
 }
    }
    if( flush_count != 0 ) {
 FlushPages( db, flushing, flush_count );
#if defined(MEASURE_WRITE_TIMES)
 flush_total += flush_count;
#endif
 flush_count = 0;
    }
    WaitAll( db_no );
    // Recompute_stats();
#if defined(MEASURE_WRITE_TIMES)
    QueryPerformanceCounter( &stop_time );
    QueryPerformanceFrequency( &frequency );
    if( flush_total > 0 ) {
 double est = (stop_time.QuadPart - start_time.QuadPart)*1000.0/(frequency.QuadPart*flush_total);
 db->_auto_ckpt->set_write_time( est );
    }
#endif
    // PJB TODO:  Figure out a better location for this.
    SetPageScoringParameters();
}
void
Cache::Flush( p_database db )
/***************************/
// Flush out dirty pages in preparation for a checkpoint.
{
    DoFlush( db, FALSE /* don't flush temp */ );
}
void
Cache::sa_flush_cache( p_database db )
/************************************/
// Implementation of sa_flush_cache internal stored procedure.
// Write all dirty pages and evict from the cache as many pages as possible
// that belong to the specified database
{
    Forbid();
    DoFlush( db, TRUE /* flush temp */ );
    a_database_number db_no = db->id();
    for( unsigned i = 0; i < avail_infos(); ++i ) {
 PageInfo * info = &_info[i];
 a_cache_name name = info->name();
 if( name.as_member.db_no == db_no && !info->IsInUse() ) {
     if( info->Latch( this, name ) ) {
  _assertD( !info->_pending && !info->_is.dirty );
  RemoveFromHash( info );
  info->Unlatch();
     }
 }
```

```
    }
    Permit();
}
void
Cache::FlushCacheForFileDrop( a_database_number db_no, unsigned file_no )
/*****************************************************************************/
// A checkpoint must be done before this call.
{
    Evict( db_no, file_no, FALSE );
#if !PRODUCTION
    CheckIfStillUsedAtFileDrop( db_no, file_no );
#endif
}
#if !PRODUCTION
void
Cache::CheckIfStillUsedAtCommit( a_database_number db_no, CacheInfo *def_page )
/*****************************************************************************/
{
    // PJB FIXME: How to avoid spurious errors?
#if 1
    _unused( db_no );
    _unused( def_page );
#else
    // Check to see if any pages are still in use.
    for( a_ptrint i=0; i < avail_infos(); i++ ) {
 PageInfo * info = &_info[i];
 if( info->db_no() == db_no
     && !_is_temporary( info->page_no() )
     && info->IsLocked()
     && info != (PageInfo *) def_page
     && info->_file->get_page_usage( info->_image ) != LOG_PAGE ) {
     cache_error( info, CACHE_ERR_STILL_LOCKED );
     // We can still trip across this when returning blobs
     // via streams:  we call CmdSeqPres::SendMultiPiece
     // with a pointer into a page.  This is likely not to
     // cause any grief (other than a potentially unbounded
     // wait while holding a page locked.)  PJB 9/14/00.
     // Another way to hit it is while adding a row to
     // a Java index; the deserialize code can call Yield(),
     // which context switches while we have locks. ITB 9 Dec 2002
 }
    }
#endif
}
#endif
a_ptrint
Cache::Clean( Database *db, unsigned rover, unsigned count )
/*****************************************************************************/
// Issue an IO for "count" dirty pages in the cache if there are
// any valid candidates.  Called by idle IO and by auto checkpoint code.
```

```
{
    a_flushed flushing[FLUSH_MAX];
    unsigned flush_count = 0;
    a_database_number db_no = db == NULL? NO_DB_ID : db->id();
    if( count > FLUSH_MAX ) {
 count = FLUSH_MAX;
    }
    a_ptrint limit = cache_available_images();
    if( limit > 500 * count ) {
 limit = 500 * count;
    }
    // Collect a set of candidates:
    unsigned i;
    for( i = rover; (unsigned) (rover - i) < limit; ++rover ) {
 // PJB FIXME:  get rid of mod.
 PageInfo * info = &_info[(rover + i)%avail_infos()];
 a_cache_name name = info->name();
 if( (name.as_member.db_no == db_no || db_no == NO_DB_ID) &&
     !_is_temporary( name.as_member.page_no ) && info->_is.dirty && !info->_pending ) {
     flushing[flush_count].info = info;
     flushing[flush_count].page_no = name.as_member.page_no;
     ++flush_count;
     if( flush_count >= count ) break;
 }
    }
    qsort( flushing, flush_count, sizeof(flushing[0]), compare_page_ids );
    for( i = 0; i < flush_count; ++i ) {
 PageInfo * info = flushing[i].info;
 if( info->TryLatch() ) {
     a_cache_name name = info->name();
     if( (name.as_member.db_no == db_no || db_no == NO_DB_ID) &&
  !_is_temporary( name.as_member.page_no ) && info->_is.dirty && !info->_pending ) {
  // PJB FIXME: _vm_protect( info );
  _checksum( info->_image );    // blank padding
  info->StartWrite()->Finish( FALSE, TRUE );
  _db_only_statistic( info->_file->db, IDLE_WRITES );
     }
     info->Unlatch();
 }
    }
    return( rover );
}
unsigned
CM_Clean( Database *db, unsigned rover, unsigned count )
/*****************************************************/
{
    // PJB FIXME:  get rid of cast.
    return( (unsigned) XM->Clean( db, rover, count ) );
}
// CacheRef operations
```

```
struct AdjustList {
    AdjustList()
  : fixup( NULL )
    {
    }
    a_page_id  page_no;
    CacheRef  *fixup;
};
IdleIOTimer::IdleIOTimer()
/************************/
    : _rover( 0 )
{
}
void
IdleIOTimer::dispatch()
/*********************/
{
#if defined( DYNAMIC_CACHE_SIZE )
    // Prevent the cache from shrinking
    PreventSuperForbid();
#endif
    _rover = CM_Clean( NULL, _rover, 1 );
#if defined( DYNAMIC_CACHE_SIZE )
    AllowSuperForbid();
#endif
}
void
Cache::adjust( DoAdjust *cb, Database *db,
        a_page_id p0, a_page_id p1, a_page_id p2 )
/*****************************************************/
{
    Worker    *me    = _CurrentWorker;
    a_database_number  db_no = db->id();
    CacheChain    *chain;
    CacheRef    *ref;
    CacheRef    *nref;
    CacheRef    **next;
    CacheChain    *bogus = me->_bogus_chain;
    AdjustList   adjust[2];
    AdjustList    *a;
    a_cache_name  name = NameFor( db_no, p0 );
    adjust[0].page_no = p1;
    adjust[1].page_no = p2;
    bogus->Latch();
    chain = ChainFor( name );
    chain->Latch();
    for( next = &chain->_refs; (ref = *next) != NULL; ) {
 if( ref->_name.as_uint == name.as_uint ) {
     uint i = cb->do_adjust( ref );
     if( i == 0 ) {
```

```
 next = &ref->_next;
   } else {
 *next = ref->_next;
 // PJB FIXME:  does this work?
 if( ref->_is.locked ) {
    // Must be on same connection.
    ref->unlock();
 }
 a = &adjust[i-1];
 ref->_info = NULL;
 ref->_name = NameFor( db_no, a->page_no );
 if( a->page_no == NULL_PAGE ) {
    ref->_chain = NULL;
 } else {
    ref->_next = a->fixup;
    a->fixup = ref;
    ref->_chain = bogus;
 }
   }
} else {
    next = &ref->_next;
}
  }
  chain->Unlatch();
  for( a = &adjust[0]; a < &adjust[2]; ++a ) {
if( a->fixup ) {
    chain = ChainFor( NameFor( db_no, a->page_no ) );
    chain->Latch();
    for( ref = a->fixup; ref != NULL; ref = nref ) {
 nref = ref->_next;
 ref->_chain = chain;
 ref->_next = chain->_refs;
 chain->_refs = ref;
    }
    chain->Unlatch();
}
  }
  bogus->Unlatch();
}
void
Cache::kick_auto_resize()
/**********************/
{
#if defined( DYNAMIC_CACHE_SIZE )
   if( _auto_resize ) {
 _auto_resize->kick();
   }
#endif
}
void
```

```
Cache::engine_startup_complete()
/*****************************/
{
#if defined( DYNAMIC_CACHE_SIZE )
    if( _auto_resize ) {
 _auto_resize->enable();
    }
#endif
}
void
Cache::cache_message( void )
/*************************/
{
    uint64  current_size = current_cache_size();
    startup_msg( IDS_ENG_CACHE_MEMORY_USAGE, (uint32)(current_size/_u64_const(1024)) );
#if defined(DYNAMIC_CACHE_SIZE)
    if( !awe_enabled() ) {
 uint64  min_size = minimum_cache_size();
 uint64  max_size = maximum_cache_size();
 startup_msg( IDS_ENG_CACHE_SIZE_RANGE,
       (uint32)(min_size/_u64_const(1024)),
       (uint32)(max_size/_u64_const(1024)) );
    }
#endif
#if defined( AWE_CACHE )
    if( awe_enabled() ) {
 startup_msg( IDS_ENG_AWE_CACHE_SIZE,
       (uint32)(_awe_cache.phys_mem_size()/(uint64)(1024)),
       (uint32)(_images.total_size()/(uint64)(1024)) );
    }
#endif
}
#if 0
// #define VM_PROTECT
typedef volatile a_ptrint a_pseudo_atomic;
#if defined( VM_PROTECT )
    #define _vm_protect( info )   XM->vm_protect( info )
    #define _vm_protect_rw( image )   XM->vm_protect_rw( image )
    #define _vm_protect_noaccess( image )  XM->vm_protect_noaccess( image )
    #define _vm_temp_protect_rw( image ) DWORD __old_access = XM->vm_protect_rw( image )
    #define _vm_restore_protection( image ) if( __old_access != PAGE_READWRITE ) XM->vm_do_protect( image,
__old_access )
    DWORD vm_do_protect( void *image, DWORD protect )
    {
 DWORD old_access;
 VirtualProtect( image, _db_page_size_max, protect, &old_access );
 return( old_access );
    }
    DWORD vm_protect_rw( void *image )
    {
```

```
 DWORD old_access;
 VirtualProtect( image, _db_page_size_max, PAGE_READWRITE, &old_access );
 return( old_access );
   }
   DWORD vm_protect_noaccess( void *image )
   {
 DWORD old_access;
 VirtualProtect( image, _db_page_size_max, PAGE_NOACCESS, &old_access );
 return( old_access );
   }
   void vm_protect( PageInfo *info )
   {
 if( info->_count || info->_pending ) {
    vm_protect_rw( info->_image );
 } else {
    vm_protect_noaccess( info->_image );
 }
   }
#else
   #define _vm_protect( info )
   #define _vm_protect_rw( image )
   #define _vm_protect_noaccess( image )
   #define _vm_temp_protect_rw( image )
   #define _vm_restore_protection( image )
#endif
#endif
void
Cache::hint_page_group( PageGroup *scatter, a_page_id start, uint32 to_hint )
/*****************************************************************************/
{
   IDatabaseFile * file = scatter->_file;
   _assertD( scatter->_ref_count == 1 );
   // "start" should be on a reasonable boundary, e.g. a multiple of num_pages.
   _assertD( (_physical_page(start) % scatter->NumBufferPages()) == 0 );
   if( _physical_page( start ) + scatter->NumBufferPages() >= file->_current_dbspace_size ) {
 // Less than a full block until end-of-file.
 return;
   }
   a_database_number db_no = file->db->id();
   IOCB * iocb = NULL;
   unsigned offset;
   unsigned length;
   for( ; to_hint != 0; to_hint >>= 1, ++start ) {
 if( (to_hint&1) ) {
    PageInfo * info = IsNotInCache( NameFor( db_no, start ) );
    if( info != NULL ) {
  if( iocb == NULL ) {
     iocb = GetIOCB( info );
     offset = 0;
  }
```

```
  iocb->_mask |= 1 << offset;
  info->PrepareForRead( iocb );
  scatter->Map( offset, info->_image );
  info->Unlock();
  length = ++offset;
  continue;
     }
 }
 if( iocb != NULL ) {
     scatter->Map( offset, NULL );
     ++offset;
 }
    }
    if( iocb != NULL ) {
 if( iocb->_mask == 1 ) {
     file->start_read( iocb->_info );
 } else {
     file->start_read( iocb->_info, scatter, length );
 }
 iocb->Unlatch();
 _db_statistic_incr( file->db, NULL, READ_HINTS );
    }
}
#if 0
#if !PRODUCTION
void Cache::_dump_cache_info( void )
/*********************************/
{
    FILE  *out;
    a_cache_index i;
    PageInfo  *info;
    out = fopen( "c:\\cache.out", "wb" );
    if( out == NULL ) {
 return;
    }
    fprintf( out, "    %-8s %-5s %-5s\n", "image", "info", "usage" );
    for( i=0; i<_config._current._n_infos; i++ ) {
#if defined(DYNAMIC_CACHE_SIZE)
 if( !awe_enabled() && !_allocation_map.test( i ) ) {
     // page is not committed
     continue;
 }
#endif
 void *image;
#if defined( AWE_CACHE )
 if( awe_enabled() ) {
     image = ordinal_to_image( i );
     if( image == NULL ) {
  break;
     }
```

```
 } else {
     image = ordinal_to_image( i );
 }
#else
 image = ordinal_to_image( i );
#endif
 info = _image_to_info( image );
 _vm_temp_protect_rw( image );
 a_byte usage = (((a_byte *)image)[4] & 0x7);
 _vm_restore_protection( image );
 fprintf( out, "IMG %08p %08p %d\n", image, info, usage );
    }
    fprintf( out, "    %-8s %-8s %-8s %5s %s\n", "info", "image", "physmem", "locks", "evicted/trashed" );
    for( i=0; i<_config._current._n_infos; i++ ) {
 info = _info.get( i );
 fprintf( out, "INF %08p %08p %08x %5d %s%s\n",
   info, info->_image, info->get_physmem_id( this ), info->_count,
   info->_is.evicted?"e":"",
   info->_is.trashed?"t":"" );
    }
    fprintf( out, "Avail Queue\n" );
    fprintf( out, "%-8s %-8s\n", "info", "image" );
    a_pseudo_atomic tail = 0;
    while( _avail.do_dequeue( tail ) != NOT_QUEUED ) {
 info = _avail.get( tail );
 fprintf( out, "%08p %08p\n", info, info->_image );
    }
    fclose( out );
}
extern "C" void Dump_cache_info( void );
void Dump_cache_info( void )
/*************************/
{
    XM->_dump_cache_info();
}
#endif
#endif
// dbpcache.h
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// ***************************************************************
// Copyright 1991-2003 iAnywhere Solutions, Inc.  All rights reserved.
// ***************************************************************
#ifndef II_DBPCACHE
#define II_DBPCACHE
// #include "utqueue.h"
// #include "atomic.h"
// #include "dbsrvapi.h"
#include "dbfile.h"
#include "dbstat.h"
class DBPage;
```

```
class IDatabasefile;
class PageInfo;
class PageGroup;
class MultiPageAlloc;
class DynamicCacheConfig;
class an_image_set;
class Cache;
class CacheRef;
class IOCB;
class CacheChain;
#define BOGUS_NAME (uint64 (~0))
union a_cache_name {
    uint64  as_uint;
    struct {
 uint32  page_no;
 uint32  db_no;
    } as_member;
};
inline a_cache_name NameFor( a_database_number db_id, a_page_id page_id ) {
    a_cache_name name;
    name.as_member.page_no = page_id;
    name.as_member.db_no = db_id;
    return( name );
}
class DBSRVAPI CacheInfo {
public:
    union a_disk_page *disk_page() {
 return( (union a_disk_page *)_image );
    }
    DBPage *db_page() {
 return( (DBPage *)_image );
    }
    IDatabaseFile *file() {
 return( _file );
    }
    Database *db() {
 return( _file->db );
    }
    void write_and_unlock( CacheInfo *delay );
    void write_and_flush();
    a_bool read_to_write_lock();
    void write_to_read_lock();
    void mark_dirty();
    void unlock();
    void Pin();
    void Unpin();
    void SetTableInfo( a_table_id tid, a_bool is_blob_page );
protected:
    friend class Cache;
    friend class CacheRef;
```

```
    friend class IOCB;
    friend class CacheChain;
    void  *_image;
    a_cache_name _name;
    IDatabaseFile *_file;
public:  // PJB TODO:  make these protected
    a_page_id page_no() const
    {
 return _name.as_member.page_no;
    }
    a_database_number db_no() const
    {
 return (a_database_number)_name.as_member.db_no;
    }
    a_cache_name name() const
    {
 return _name;
    }
};
class DoAdjust {
public:
    virtual a_uint do_adjust( CacheRef *ref ) = 0;
};
unsigned CM_Clean( Database *db, unsigned rover, unsigned count );
#if !defined( DBTOOLS )
DBSRVAPI a_database_file *file_for_page( p_database db, a_page_id pg );
DBSRVAPI a_database_file *check_file_for_page( p_database db, a_page_id pg );
#endif
static inline unsigned log2( a_ptrint n ) {
    unsigned l;
    for( l = 0; ((a_ptrint)1 << l) < n; ++l );
    return( l );
}
class CacheInterface {
public:
    CacheInterface( struct an_engine_parms * ep );
    virtual ~CacheInterface();
    virtual void *AllocImage( CacheInfo ** ) = 0;
    virtual void FreeImage( CacheInfo * info ) = 0;
    virtual CacheInfo * Install( a_database_file * f, a_page_id page_no, a_page_id real_page_no ) = 0;
    virtual void WaitAll( a_database_number db_no ) = 0;
    virtual void WaitForExtend( PageInfo * extend ) = 0;
    virtual void WaitUntilClean( IDatabaseFile * file ) = 0;
    virtual a_bool RemoveFromHash( PageInfo * info ) = 0;
    virtual PageInfo * LatchIfInCache( a_cache_name name ) = 0;
    virtual a_bool IsImmediatelyLatchable( a_cache_name name ) = 0;
    virtual a_bool IsIOPending( a_cache_name name ) = 0;
    virtual void PreIO() = 0;
    virtual void PostIO() = 0;
    virtual void evict( Database *db, a_page_id page_no ) = 0;
```

```
    virtual unsigned hint( Database *db, a_page_id page_no ) = 0;
#if !PRODUCTION
    virtual void check_for_locked_pages( int k ) = 0;
#endif
    virtual a_ptrint cache_available_addr( void ) = 0;
    virtual a_ptrint __virtual__cache_max() = 0;
    virtual void flush_db_from_cache( p_database db, a_bool is_scrammed ) = 0;
    virtual void flush_mapped_pages( p_database db ) = 0;
    virtual void flush_cache_for_table_drop( p_database db, unsigned file_no ) = 0;
    virtual void flush( p_database db ) = 0;
    virtual void sa_flush_cache( p_database db ) = 0;
    virtual void adjust(  DoAdjust *cb, Database *db, a_page_id p1, a_page_id p2 = NULL_PAGE, a_page_id p3 =
NULL_PAGE ) = 0;
    virtual unsigned hint( a_page_id page_no ) = 0;
    virtual void hint_page_group( PageGroup *pagegrp, a_page_id start_page, uint32 pages_wanted ) = 0;
    virtual void *multi_page_alloc( uint32 num_pages, uint32 page_size, MultiPageAlloc *handle ) = 0;
    virtual void multi_page_free( MultiPageAlloc *handle ) = 0;
    virtual void cache_message() = 0;
    virtual void engine_startup_complete() = 0;
    virtual void kick_auto_resize() = 0;
    virtual DynamicCacheConfig * min_config() = 0;
    virtual DynamicCacheConfig * max_config() = 0;
    virtual DynamicCacheConfig * current_config() = 0;
    virtual DynamicCacheConfig * initial_config() = 0;
    virtual a_bool resize_cache( uint64 newsize ) = 0;
    virtual an_image_set * get_images() = 0;
    unsigned page_bits_max() const {
  return( _db_page_bits_max );
    }
    unsigned page_size_max() const {
  return( _db_page_size_max );
    }
    a_ptrint page_mask_max() const {
  return( _db_page_mask_max );
    }
    uint64 aligned_size( uint64 size ) const {
  return( _round_up_pow2( size, _alignment_amount ) );
    }
    a_ptrint enough_to_fill_pages( double min_count, a_ptrint item_size ) const {
  return( (a_ptrint) (aligned_size( ((a_ptrint) min_count) * item_size )/item_size) );
    }
    a_ptrint round_up_ordinal( a_ptrint ordinal ) const {
  return( _round_up_pow2( ordinal, _ordinal_alignment ) );
    }
    a_ptrint round_down_ordinal( a_ptrint ordinal ) const {
  return( _round_down_pow2( ordinal, _ordinal_alignment ) );
    }
protected:
    friend class AWECache;
    unsigned const _db_page_bits_max;
```

```
    unsigned const _db_page_size_max;
    a_ptrint const _db_page_mask_max;
    a_ptrint const _os_page_size;
    unsigned const _os_page_bits;
    a_ptrint const _alignment_amount;
    a_ptrint const _ordinal_alignment;
public:
    virtual CacheInfo *read_lock( p_database db, a_page_id page_no ) = 0;
    // I don't like referring to 'a_disk_page' at this level but it helps with type checking
    CacheInfo *read_lock( p_database db, a_page_id page_no, union a_disk_page **ppage )
{
     CacheInfo *info = read_lock( db, page_no );
     *ppage = info->disk_page();
     return( info );
}
    virtual CacheInfo *write_lock( p_database db, a_page_id page_no ) = 0;
    CacheInfo *write_lock( p_database db, a_page_id page_no, union a_disk_page **ppage )
{
     CacheInfo *info = write_lock( db, page_no );
     *ppage = info->disk_page();
     return( info );
}
    virtual CacheInfo *write_lock_clean( p_database db, a_page_id page_no ) = 0;
    CacheInfo *write_lock_clean( p_database db, a_page_id page_no, union a_disk_page **ppage )
{
     CacheInfo *info = write_lock_clean( db, page_no );
     *ppage = info->disk_page();
     return( info );
}
    virtual CacheInfo *write_lock_raw( p_database db, a_page_id page_no ) = 0;
    a_bool read_to_write_lock( CacheInfo ** pinfo ) {
 CacheInfo * info = *pinfo;
 if( read_to_write_lock( info ) ) {
     return( TRUE );
 }
 a_page_id page_no = info->page_no();
 p_database db = info->file()->db;
 unlock( info );
 *pinfo = write_lock( db, page_no );
 return( FALSE );
    }
    a_bool read_to_write_lock( CacheInfo ** pinfo, union a_disk_page **ppage ) {
 if( read_to_write_lock( pinfo ) ) {
     return( TRUE );
 }
 *ppage = (*pinfo)->disk_page();
 return( FALSE );
    }
public: // PJB TODO:  Eliminate these wrappers.
    a_bool read_to_write_lock( CacheInfo *info ) {
```

```
 return info->read_to_write_lock();
    }
    void write_to_read_lock( CacheInfo *info ) {
 info->write_to_read_lock();
    }
    void mark_dirty( CacheInfo *info ) {
 info->mark_dirty();
    }
    void unlock( CacheInfo *info ) {
 info->unlock();
    }
#if !PRODUCTION
    virtual void check_if_still_used_at_commit( p_database db, CacheInfo *def_page ) = 0;
#endif
};
extern CacheInterface *CM;
class AutoImage
{
public:
    AutoImage()
 {
     CM->AllocImage( &_info );
 }
    ~AutoImage()
 {
     release();
 }
    CacheInfo *info()
 {
     return( _info );
 }
    void *image()
 {
     return( (void *)_info->disk_page() );
 }
    union a_disk_page *disk_page()
 {
     return( _info->disk_page() );
 }
    void release()
 {
     if( _info != NULL ) {
  CM->FreeImage( _info );
  _info = NULL;
     }
 }
private:
    CacheInfo *_info;
};
#endif
```